Soft Computing Optimizer (SCOptimizer)

Version 1.1

User Guide

1 Contents

2 Before you begin

This user guide assumes that you are familiar with fuzzy control theory and genetic optimization algorithms.
`Courier font` indicates program interface objects, like menu items, windows, pages and so on.
`BOLD Courier` indicates words or characters you type.
Important terms are selected with **bold** font.

2.1 Structure of this guide.

Section **3 Introduction to Soft Computing Optimizer** tells you about the process of model creation with SCOptimizer. It will show you which steps you should perform and what is required for these steps.

Section **4 Using Soft Computing Optimizer** is devoted to work with SCOptimizer interface. First part of this section **Starting with Soft Computing Optimizer** describes basic interface principles**. Working with SCOptimizer** tells you how to use SCOptimizer to perform different optimization tasks and how to view and edit your model.

Section **5 Using Inference tells** you about Inference application, stand-alone tool designed to simulate a fuzzy system behavior.

Section **6 Using stand alone inference library from C++ code** shows an example on how you can use models created in SCOptimizer in your C/C++ program. For more details on this topic consult SCLib reference.

Reference information of different kinds is available in Appendixes.

3 Introduction to Soft Computing Optimizer

Soft Computing Optimizer (SCOptimizer) is a software tool for the automatic fuzzy model generation. SCOptimizer uses samples of Input-Output vectors to create and optimize model of fuzzy system.

SCOptimizer design flow consists of following main steps:

1. Project creation
2. MF's shape definition
3. Rule database creation
4. Rule database optimization
5. Fine tuning of the model

SCOptimizer is a Win 32 application. Supported operating systems are Windows 2000 and Windows XP. For its functionality it requires any AMD or Intel CPU based PC with at least 128MB of memory and 10MB of hard disk space for program and swap files. Matlab based optimization can be used with Matlab version 6.0 or higher.

3.1 Project creation

The first menu of SCOptimizer will allow you to create a new model or load previously created model from file. If you choose to create a new model the system will prompt you about model parameters, including inference model, number of input and output variables, number of fuzzy sets for each variable and so on.

After the model was created or loaded you will be presented with main program menu, allowing you to view model parameters, start different optimization algorithms or edit model manually.

3.2 Membership function's shape definition and optimization

The first step of model optimization is the definition of shape of membership functions of fuzzy sets of input and (if used by the model) of output variables. SCOptimizer supports two modes of MF's shape definition: using **uniform distribution** method or with **GA1** optimization algorithm.

Uniform distribution method distributes fuzzy sets on signal change interval according to signal probability distribution and user selected shape of membership functions.

GA1 algorithm tries to find best possible combination of number of fuzzy sets per variable, membership function shape and overlap coefficient between neighbor fuzzy sets. For each combination it performs uniform distribution algorithm and tries to maximize the mutual possibility of the fuzzy sets of each variable.

3.3 Rule database creation

The main part of the model is a rule database. It stores data, which shows which output should be activated for given input. SCOptimizer supports two types of rule database: **complete database** and **LBRW database**.

Rules of **complete database** present all possible combinations of fuzzy sets of input variables. Number of rules in complete database equals to product of numbers of fuzzy sets of input variables. This will result in extremely large database and very slow optimization speed if you will try to use it with more than one-two variables.

LBRW database store not all the rules, but only a number of rules selected with "Let the Best Rule Win" algorithm. **LBRW algorithm** selects those rules, which contribute the most noticeable part of the output. Reducing number of rules with **LBRW algorithm** provides faster optimization speed without loss of model precision.

3.4 Rule database optimization

After the database was created it should be filled with actual rule data. This is accomplished on the final step of model creation – rule database optimization. SCOptimizer uses genetic optimization algorithm (**GA2**) to tune database parameters.

3.5 Fine tuning of the model

Quality of the model created during previous steps may still be inadequate. In order to improve model quality **GA3** algorithm is used. It alters shapes of membership functions and optimizes model output

with fixed number of membership functions and database structure. **Error back propagation** algorithm can be used to improve model output but fine-tuning database parameters using classical gradient optimization method.

3.6 Teaching signal for SCOptimizer

In order to perform different optimization algorithms SCOptimizer requires **teaching signal**, which presents samples of input values and corresponding output values. SCOptimizer is able to read signal data from Matlab v.4 and v.5 files and from text files.

Text files are processed based on **locale data**, which defines symbols for decimal point, thousands separators and so on. By default SCOptimizer uses windows settings for these parameters. If those settings do not match signal file format they can be changed at any moment. Once changed, locale parameters are saved in model and will be used for future processing of data. Locale setting affects reading and writing of text data files and model files.

4 Using Soft Computing Optimizer

4.1 Starting with Soft Computing Optimizer

Main module of Soft Computing Optimizer is a Win32 application, called scowin.exe. It can be used in Windows 95, 98, NT, Me, 2000 and XP environments.
In order to start the program you should type “SCOptimizer” in command prompt or double-click SCOptimizer icon in windows explorer.
During process of model optimization SCOptimizer creates temporary files in current directory. Those files have “.mms” extensions. Before starting SCOptimizer check that current directory does not contain files with same extension.

4.1.1 SCOptimizer program window

Main program window of SCOptimizer is shown on the figure 4.1.1.1.

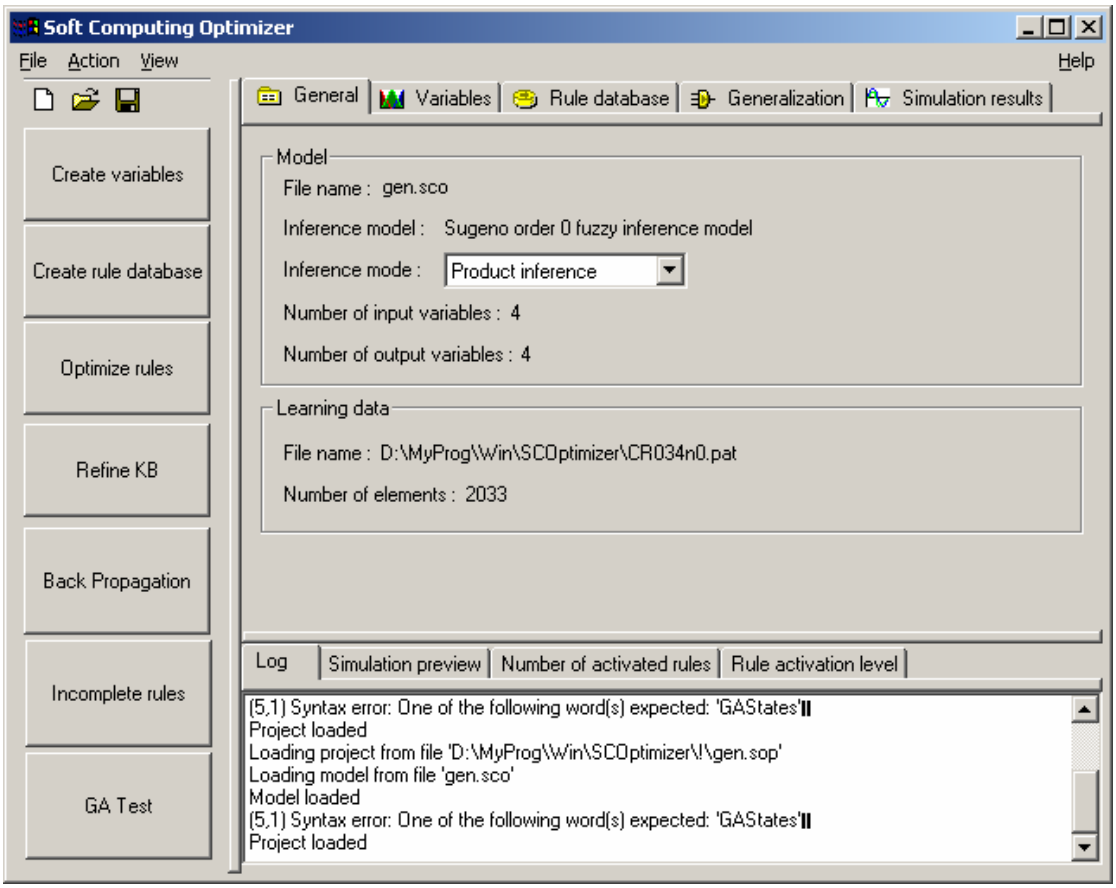

Figure 4.1.1.1. SCOptimizer program window.

SCOptimizer window is dived into three parts. On the left part of the window there are command buttons, which activates main design steps. By using those buttons you will be able to proceed with creation of model from the first step of MF shapes to the final refinement of model. Initially some of the buttons are disabled, because actions performed by those buttons cannot be performed. Those buttons will become available after completing previous steps.

Right part of the window is devoted to displaying model parameters. It is organized as pages, each of which displays different properties of the model. You can switch between pages using tabs. Top part consists of five pages: `General`, `Variables`,`Rule database`,`Generalization` and `Simulation results`. `General` page display main model parameters including file names, inference model and so on. `Variables` page displays input and output variables and their membership functions. You can use this page to manually edit membership functions. `Rule`

database page reflects state of model rule database. Generalization page allow you to view and change generalization parameters, if you model consists of several databases. Simulation result page displays graphics of model output.

Bottom part of the window has four pages. The Log page contains model creation log. SCOptimizer prints all messages during model creation in this window. Other pages display various graphs which can be used to determine model quality. Simulation preview page displays teaching signal and model output. Graphs on this page are immediately updated after you change any model parameter. Number of activation rules page displays graph with number of rules activated for each teaching signal line. Rule activation level page displays graph of maximal rule activation level for teaching signal.

4.1.2 SCOptimizer toolbar

SCOptimizer toolbar is located atop of command buttons. Three buttons on this toolbar provide fast access to the FILE menu functions:New, Open and Save. See section 4.1.3.1 for information about those functions.

4.1.3 SCOptimizer menu

SCOptimizer also has standard window menu, displayed on the top of program window. To choose a command with the mouse or keyboard, first select the menu and then choose command you want. Each underlined character in menu and command names corresponds to the key you can press to select a menu or to choose a command.

To choose a command by using the mouse you should first click the menu name containing the command you want and then click the command name in the pop-up menu.

To choose a command by using the keyboard first press the **ALT** key. The File menu appears selected. Use left and right arrow keys to select required menu and then press down arrow or **ENTER** to open pop-up menu. Use up and down arrows to select desired command and press **ENTER** to activate it. You can also press one of the underlined characters to quickly select desired menu or command.

4.1.3.1 FILE Menu

SCOptimizer allows project files to be managed by the means of commands in the FILE Menu.

The following items are available in this menu:

New: closes actual model and starts creation of new model.
Open: open existing model from disk file.
Load Teaching Signal: select file with teaching signal.
Save: save actual model to the file it was loaded from.
Save As: save actual model to another file.
Export: export model to another format. Currently only SCOptimizer v.1 format can be exported.
Number Format: set number format conventions for teaching signal.
Exit: closes SCOptimizer.

4.1.3.2 ACTION Menu

Action menu contains commands identical to command buttons. The following commands are available:

Generate Variables: create variables using Uniform distribution or GA-1 algorithms.
Create Rule Database: create rule database.
Optimize Rule Database: optimize rule database using GA-2 algorithm.

`Refine KB`: optimize model with `GA-3` algorithm.
`Back propagation`: optimize rule database with back propagation algorithm.
`GA Test`: run abstract genetic optimization.

4.1.3.3 VIEW Menu

View menu can be used to switch between SCOptimizer pages:

`Project properties:` display `General` properties page.
`Variables`: display `Variables` page.
`Rule Database`: display `Rule Database` page.
`Generalization`: display `Generalization` page.
`Simulation Results`: display `Simulation Results` page.
`Log`: display `Log` page.
`Simulation preview`: display `Simulation preview` page.
`Number of activated rules`: display `Number of activated rules` page.
`Rule activation level`: display `Rule activation level` page.

4.1.3.4 HELP Menu

`Help` menu currently have only one command – `About`, which displays version information of SCOptimizer.

4.1.4 Dialog boxes

When you choose a command having options, SCOptimizer shows a dialog box. A dialog box may contain fields in which you can enter text, numbers or select some items. Typical dialog box will have `OK` button, which you should press after filling all fields in order to activate the command, and `CANCEL` button, which will abort command execution. You can also use `ENTER` key on the keyboard to activate the same function as `OK` button and `ESC` as `CANCEL`.

If an operation require many parameters Wizard-style dialog box will be used. Wizard box consists of a sequence of dialog boxes that will guide you through the steps of an operation. Three buttons will be available for navigation of the wizard box. `NEXT>>` button tells wizard that you have successfully completed filling of current page and want to switch to the next page or perform the command if no additional parameters are required. `<<BACK` button should be used to return to one of the previous pages, if you want to change some parameters. `CANCEL` button aborts command execution.

4.1.5 SCOptimizer files

SCOptimizer saves project data in two files: **model file** (with `.sco` extension) and **project file** (`.sop` extension). When you save or load SCOptimizer model it always saves or loads both files simultaneously.

Model file is used to store model data, including model type, variables, their membership functions and rule database.

Project file contains information about files used in the project, including **model file** and **teaching signal** file.

SCOptimizer can save genetic algorithm optimization state to disk file, so you can continue optimization process interrupted some time ago. Those files have `.st` extension and will be called **state file**.

If you wish to copy SCOptimizer files to new location (for example to another computer) you should copy **model file, project file, teaching signal** file and **state files**, if it is required. If your **teaching signal** file was originally located in a directory other than one of the project file, then SCOptimizer may be unable to load it automatically in the new location. In this situation use `File/Load`

`Teaching Signal` command to point SCOptimizer to new location of **teaching signal** file. See section 4.2.4 about details on loading teaching signal.

4.2 Working with SCOptimizer

4.2.1 Creating new model

Select File/New from menu to start creation of the model. Wizard box will appear which will guide you through the creation of new model. When you create a new model you current model (if any) is closed. Please save you changes before creating new model.

On the first page of the dialog you should enter following parameters:

- `Model file name`: name of the file of the model. You can use browse (…) button to pick up the file name on `File Save` dialog.
- `Inference model`: fuzzy inference model. SCOptimizer supports three inference models: **Mamdani model** and **Sugeno order 0** and **1** model.
- `Inference mode`: select type of operation for fuzzy OR. This can be **product** or **minimum**.

After filling those parameters press `NEXT>>` to switch to the next page.

Select required number of input and output variables and press `NEXT>>` again.

Next two pages collect properties of input and output variables. Each page contains list with suggested variable names and number of membership functions. To change any of those parameters select line from the list. Variable name and number of MF's will appear in the fields below the list. Alter those parameters how you like and press `SET` button to apply changes.

When you are done press `NEXT>>` to switch to the next page.

Last page of this dialog allows you to select **teaching signal** file. Enter file in the field labeled `Teaching signal file name` or press browse (…) button to pick file with `File Open` dialog. If you use text format of the **teaching signal** file check that current **locale settings** displayed on this page match you file format. If those settings do not match press `Change…` button to change them. See section 4.2.5 for details about **locale settings**.

After you press `NEXT>>` on this page new model will be created. Messages describing creation of the new model will be displayed on the `Log` page.

4.2.2 Loading model from file

If you have a file with previously saved model you should select `File/Open` command. Standard windows `File Open` dialog box will appear. Select file of your model (whether **model file** or **project file**) and press `Open` button.

SCOptimizer will load selected **model file, project file** and corresponding **teaching signal** file, if available. Additional messages describing which files were loaded or error messages will be printed on the `Log` page.

4.2.3 Saving model files

You can use `File/Save` command to save you model files under the name shown on the `General` properties page. If you want to use another file name choose `File/Save As` command. Standard windows `File Save` dialog box will appear. Select file of your model (whether **model file** or **project file**) and press `Save` button.

We recommend saving your model regularly, in order to avoid loss of data due to possible program/computer failure or undesired model change. Note that most operations of SCOptimizer do not have “undo” option, so the only way to restore to previous state is to load model from saved file.

4.2.4 Loading Teaching Signal

If you want to change current **teaching signal** or if teaching signal file was not loaded before, you should select `File/Load Teaching Signal` command.

SCOptimizer may load teaching signal from Matlab v.4 and v.5 files and from text files. File type is autodetected. If the file does not look like Matlab file then it assumed to be a text file.

Matlab file should contain an array of real numbers with number of columns equal to sum of number of input and output variables of the model. If version file contains several variables then the first one will be loaded.

Text files are processed using **locale settings**. Please set **locale settings** as described in section 4.2.5 before loading text files. If **locale settings** do not match the file format then signal will be loaded incorrectly and no error message will typically be displayed. Text file should contain **teaching signal** data separated by any separators. This will include text files produced by Matlab, CSV files and practically any other file of this kind.

After selecting `File/Load Teaching Signal` command SCOptimizer will display `Open File` dialog box, where you should select the file with **teaching signal**. After selecting the file press `Open` to load it. Additional messages may be displayed on the `Log` page.

4.2.5 Changing locale settings

This operation is available from `File/Number Format` menu and from New Model creation Wizard. After selection of this operation SCOptimizer will show you dialog box, containing current settings, like those:

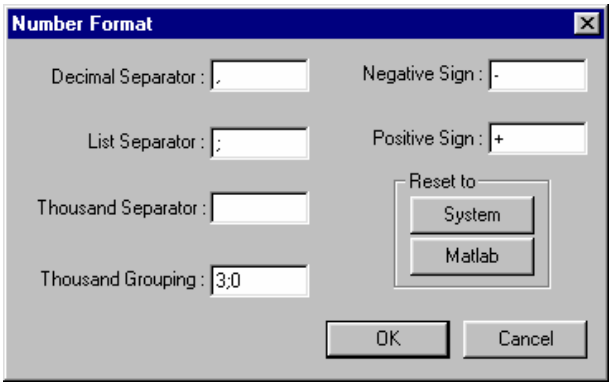

Figure 4.2.5.1. Number Format dialog box.

Every string in this list shows one of the locale parameters. Those parameters have following meaning:

List separator

List separator is a symbol or string used to separate several successive numbers in list.

Decimal separator

Separates integer and fractional part of numbers.

Negative sign

Defines negative numbers.

Positive sign

Shows that number is positive. SCOptimizer won’t print this string before positive numbers, but will accept numbers with this sign in data files.

Grouping method

Define how digits of integer part of numbers will be grouped. This is a semicolon-separated list of numbers, defining number of digits in each group, from right to left. Trailing 0 means "use previous value for all following groups". SCOptimizer will insert the thousand separators in every place defined by grouping method, but will accept thousand separators in any position in input files.

Thousand's separator

Separates groups of digits, defined by **Grouping method**.

You can change any parameter by entering desired symbol(s) to the corresponding field. There are also two buttons, which you can use to switch to one of the predefined configurations:

`System`: switch to system defaults, as defined by Windows locale settings.
`Matlab`: switch to format, used by Matlab for text files. Following settings are default for the Matlab:

```
List separator: ';'
Decimal separator: '.'
Negative sign: '-'
Positive sign: '+'
Grouping method: '3;'
Thousand's separator: ' '
```

When entering locale parameters be careful not to:
- set different parameters to the same symbol
- make list separator, decimal separator or negative sign an empty string
- set any parameter to one of the following symbols: **(**,**)**,**{**,**}**,:,'.

Breaking these rules may cause a program not to save and/or load files correctly.

After you set all options to desired characters press `OK` to apply changes or `CANCEL` if you want to reset your changes.

4.2.6 Viewing and editing the Model Parameters

The first page of model parameters you see is the `General` page. It contains project information including:

- name of the file of model
- inference model
- inference mode (operation used as fuzzy OR)
- number of input and output variables
- name of the file of **teaching signal**
- number of **teaching signal** samples in file

Inference mode may be changed by selecting minimum or product inference mode from drop-down list. Other parameters in this window cannot be changed.

To get detailed information about model variables and rule database switch to corresponding pages, described in the following sections.

4.2.6.1 Working with Variables

Model **variables** can be viewed and edited on the `Variables` page. You can switch to this page by clicking on the Variables tab or by selecting `View/Variables` menu command.

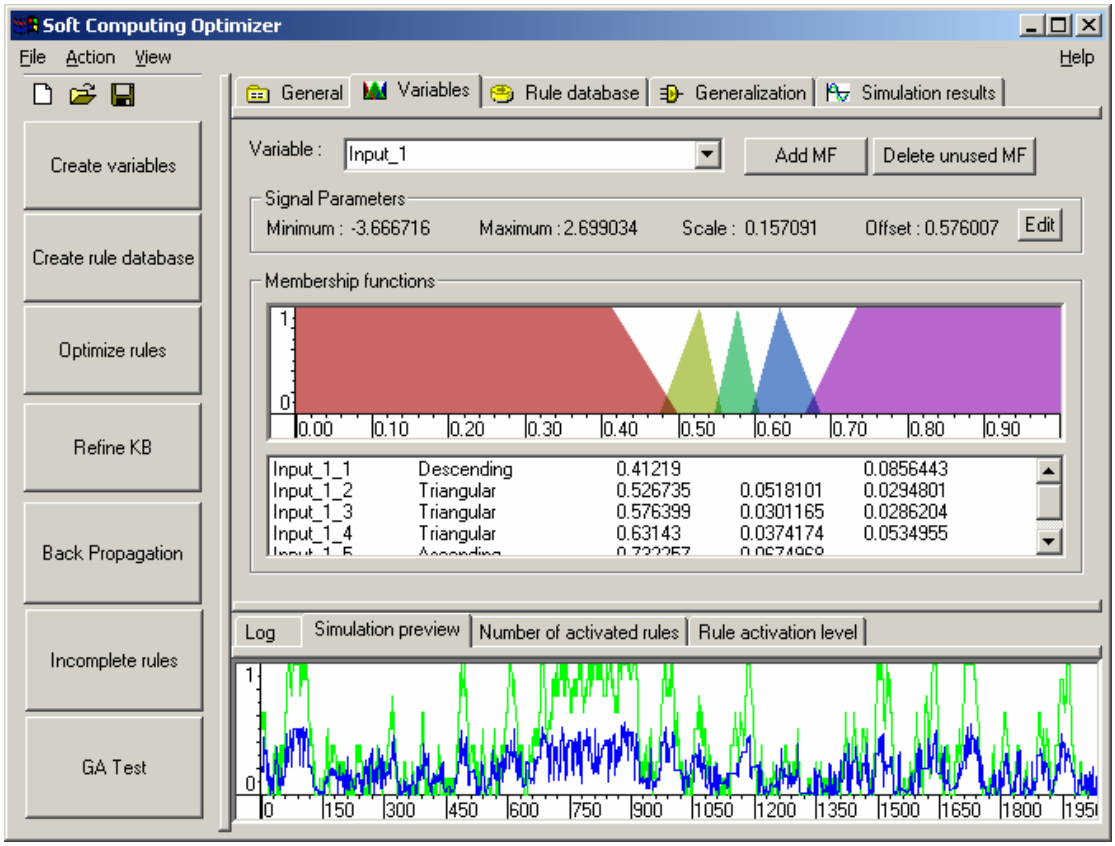

Figure 4.2.6.1.1. Variables page.

There is the name of the current **variable** at the top of the list. You can switch to another variable by selecting variable name from the drop-down list.

Then parameters of the signal are listed. `Minimum` and `maximum` are margins of signal change interval. SCOptimizer uses normalized signal for internal calculations. Normalization parameters are `Scale` and `Offset`. The following formula is used for normalization:

normalized_value = (input_value-OFFset)*`Scale`.

The rest of the `Variables` page display **membership functions** (MF's) of the variable.

Graphical window display **distribution functions** of MF's. You can change appearance of this window by the pop-up menu, activated by the right-click of the mouse in the window. Menu items are the following:

`Denormalized space`: Draw x-axis coordinates using **denormalized** (signal) space.

`Normalized space`: Draw x-axis coordinates using **normalized** (internal for SCOptimizer) space.

`Track cursor`: Use this feature to see the margins of alpha-levels. When mouse cursor is on the y-axis then lines representing alpha level will be drawn, as well as color lines which will show margins of this level for all MF's. When cursor is somewhere else on the window then vertical lines at the position of cursor and horizontal lines from intersections of this line with MF's will be drawn.

`Display MF supports`: Display supports of MF's using colored vertical lines.

`Display signal interval`: Display margins of signal change interval with vertical lines.

`Color shapes`: Draw functions using filled color figures (default).

`Color lines`: Draw functions using color lines.

`B&W lines`: Draw functions using black lines.

`Save Image`: Save current image to file (Windows BMP format).

You can use this window to change MF distribution parameters. Move mouse to the x coordinate of the modal value or support margin of one of the MF's. Colored line will appear showing selected

parameter. Press and hold left mouse button. Move mouse left or right to change the parameter. New shape of the MF will be drawn using outline method. Release left mouse button when you are satisfied with the shape of MF.

The list in the very bottom of the page display **membership functions** and their parameters. First column of the list is MF name, next – **distribution type** followed by **distribution parameters**.

You can change those parameters by double clicking list items. If you do, the following dialog box will appear:

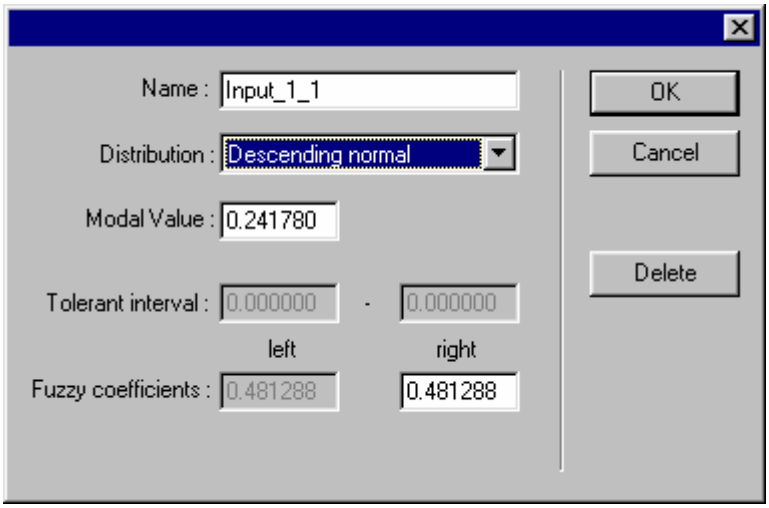

Figure 4.2.6.1.2. Membership function parameters dialog.

This dialog box displays all parameters of the membership function. Parameters can be changed by entering new data into corresponding fields. Please note that some parameters may not be available for different distributions. For example **Descending normal** distribution (as displayed on the figure 4.2.6.1.2) require only **Modal Value** parameter and **Right Fuzzy Coefficient** parameter. Fields, corresponding to the unused parameters, are grayed and can not be changed.

When you have done with this dialog press `OK` to apply you changes or `Cancel` to return without modifying the variable.

You can also use this dialog to delete **membership functions** of input variables. If you wish to do it press `Delete` button. If the **rule database** is already created then all rules with **if-part** using this membership function will also be deleted.

`Add MF` button allow you to manually add new membership function to the current variable. After you press this button the same dialog as was used for editing new variable will appear and you will be able to enter parameters of newly added membership function. If you press `OK` MF with those parameters will be added, if you press `CANCEL` it won't. Since rule database structure is highly dependant on number of MF's, rule database will be cleared and recreated if you add a MF.

`Delete unused MF` button deletes those MF's of current variable which are not used by the database, that is those MF's that are not listed in left part of the rules.

On this page you can also change **signal range** parameters of the variable. To do this press `Edit` button inside `Signal parameters` box. Following dialog box will appear:

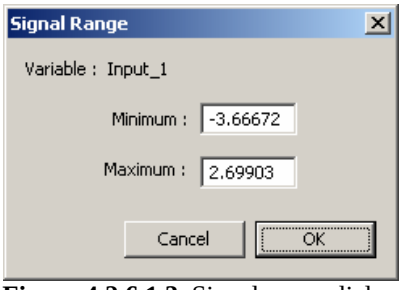

Figure 4.2.6.1.3. Signal range dialog

Enter required signal range margins into Minimum and Maximum fields and press `OK` to apply changes.

4.2.6.2 Working with Rule Database

To view or edit **rule database** you should switch to `Rule Database` page. Do this by clicking on the Rule database tab or by selecting `View/Rule Database` menu command.

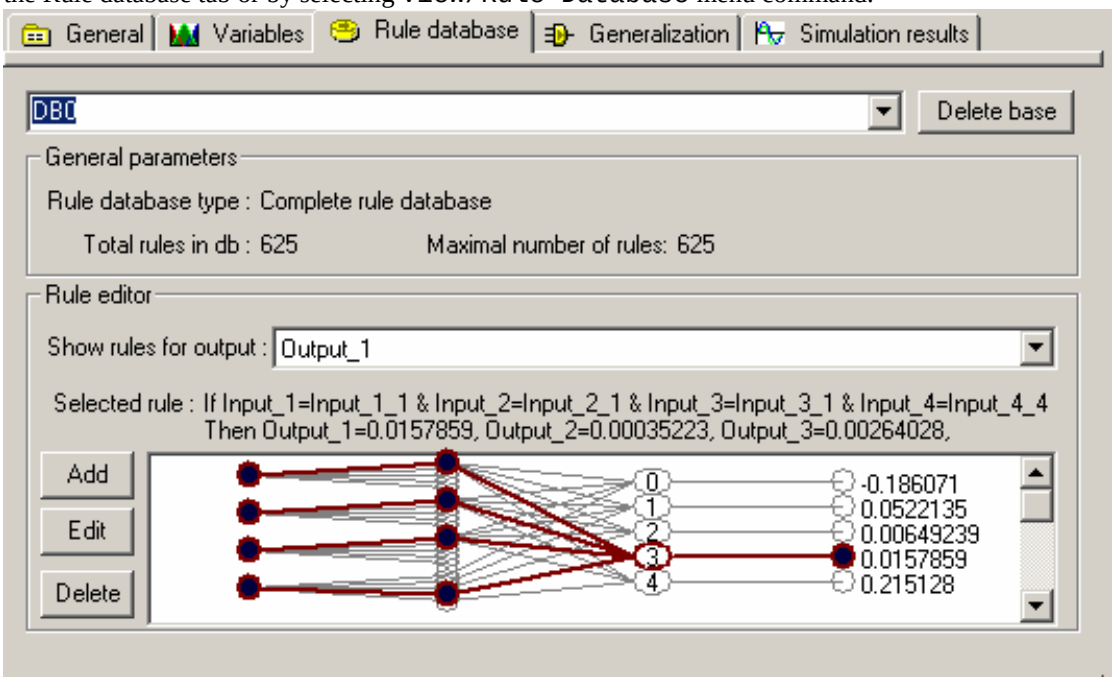

Figure 4.2.6.2.1. Rule database page.

This page displays following parameters:

`Rule database type`: type of the database used in model. Can be **Complete rule database** or **LBRW Rule database**.

`Maximal number of rules`: Maximal number of rules for current model.

`Total rules in db`: Number of rules stored in the database. For **complete database** this is always equal to maximal number of rules. For **LBRW database** it can be less then maximal.

`Show rules for output`: Rule database is displayed for one of the outputs. This list selects output variable for which database will be displayed.

`Selected rule`: Displays textual representation of selected rule, if any.

Combo box in the top of the page displays name of currently selected database. You can change database by selecting it from the drop-down list. `Delete base` button to the left of the combo box can be used to delete selected database.

Rule Database editor display database as network with four layers. First layer `is input variable layer`. Each circle in this layer represents **input variable**. Second layer is `input MF layer`. Circles of this layer represent **membership functions** of variables. Circles in the third layer represent

rules of the database. Number written inside this circle is a **rule number** of the rule in the database. Last layer is the `output layer`. For **Mamdani model** `output layer` is composed of circles, corresponding **to membership functions** of selected output variable. For **Sugeno models** `output layer` displays numerical parameters of the rule.

Database structure is shown with lines, connecting different layers. Each node in the `rule level` is linked with those MF's in `input MF layer`, which are included in the **if-part** of the rule. It is also linked with output MF or numerical parameter of the **then-part**.

Since all rules of the model won't fit on display, scroll bar is implemented which scrolls nodes of the `rule level`.

You can select a **rule** from the database by clicking on the node of the `rule level`. Textual representation of the rule will be shown in the `Selected rule` field and you will be able to edit or delete this rule. When you select a rule in the database it activation level is displayed as red lines on the `Rule Activation Level` page.

If you wish to delete the rule, select it and press `delete` button. Rule will be removed from the database. Note that rules from complete database can not be deleted.

By pressing `Edit` button you can change selected rule. The following dialog will appear:

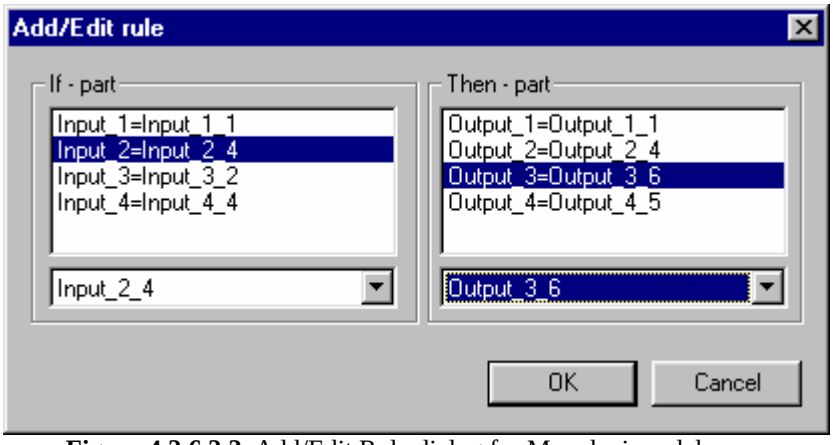

Figure 4.2.6.2.2. Add/Edit Rule dialog for Mamdani model.

The left part of the dialog represents **if-part** of the rule, and the right part corresponds to **the then-part**. You can change parameters of any part by selecting items from the list and changing values in the drop-down box below the list.

For **Sugeno 0** model this dialog will have a slightly different appearance:

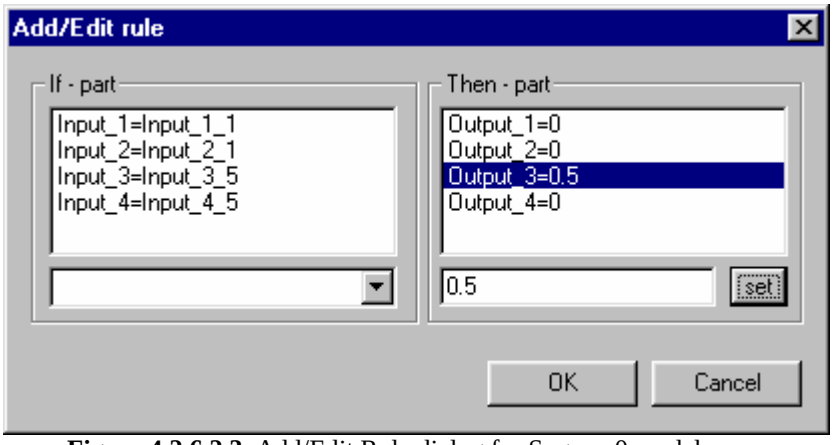

Figure 4.2.6.2.3. Add/Edit Rule dialog for Sugeno 0 model.

To change output parameter for **Sugeno 0** model select corresponding line from the list, enter new value in the text field below and press `set`.

You can add rule to the database by pressing `Add` button. The same dialog as used for rule editing will appear. Change values in the **if-part** and **then-part** as desired and press `OK` to add rule. If the rule with selected **if-part** already exists in the database it will be replaced with new rule.

4.2.7 Generalization Page

Generalization page can be used to view and control current generalization mode. Activate it by clicking the mouse on the `Generalization` tab or by selecting `View/Generalization` menu command.

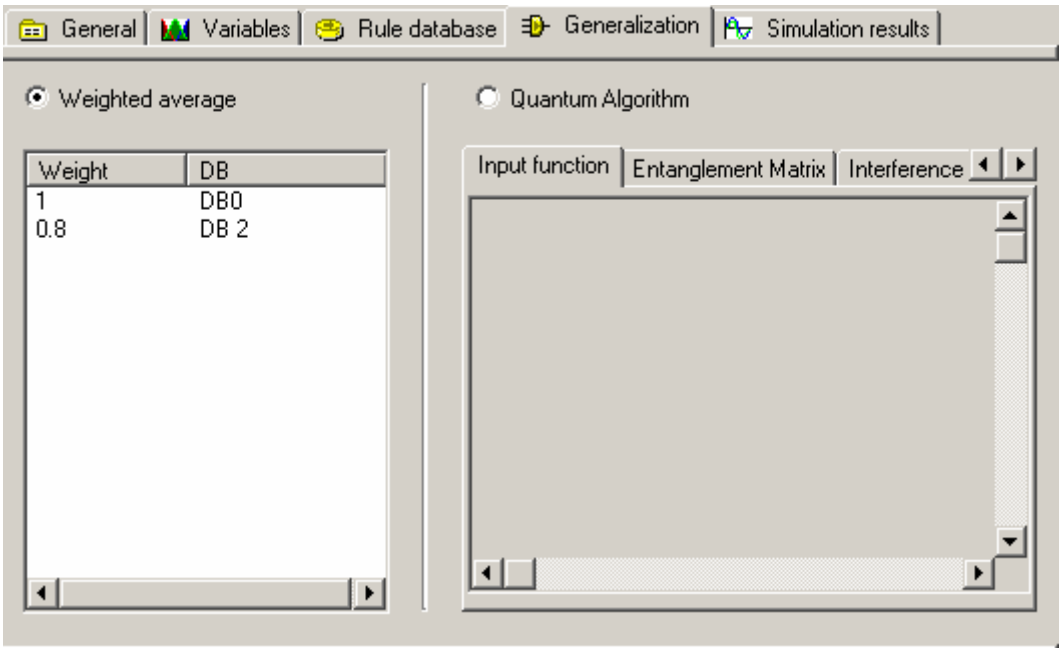

Figure 4.2.7.1. Generalization page – Weighted average mode.

Generalization algorithm determine how inference results from different rule databases are combined together to get final result. Currently two modes are available: **weighted average** and **quantum algorithm**. You can change mode by switching corresponding radio buttons on this page.

In **weighted average** mode inference result from each base is multiplied to some value (known as **weight**) and than added together and normalized (so that sum of all weights equals 1). To change the weight click on the corresponding string in the list and enter new value.

In **quantum algorithm** mode output is defined by quantum algorithm based on input function. When `Quantum algorithm` radio button is selected, control, which is responsible for displaying input function is activated. Input function is displayed as a table. Grayed fields represent function input and cannot be changed, white fields represent function value and can be toggled by clicking on them with the mouse.

You can also see graphical representation of quantum algorithm matrixes resulting from input function by clicking `Entanglement matrix` and `Inference matrix` tabs. Matrioxes are displayed as tables, in which color represent value of corresponding element. Red color corresponds to positive value, blue color – negative and white value of zero. Solid color corresponds to absolute value of 1 and color intensity is reduced together with absolute value of element.

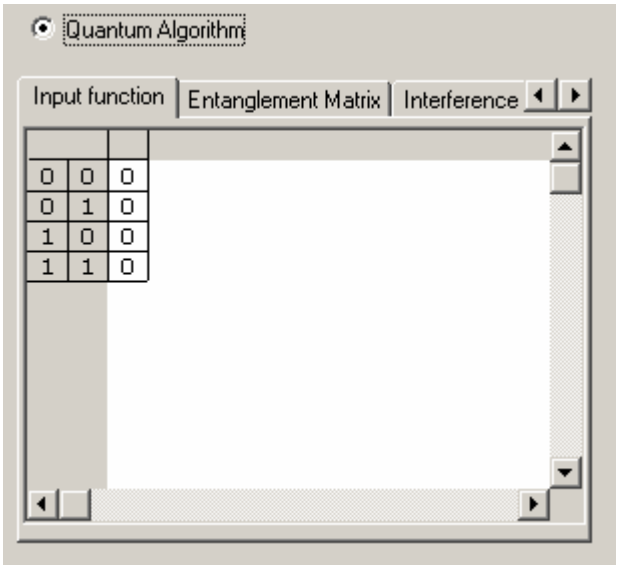

Figure 4.2.7.2. Generalization page – Quantum algorithm mode.

4.2.8 Simulation Results Page

Simulation results page can be used to verify current model output and quality. Activate it by clicking the mouse on the `Simulation results` tab or by selecting `View/Simulation results` menu command.

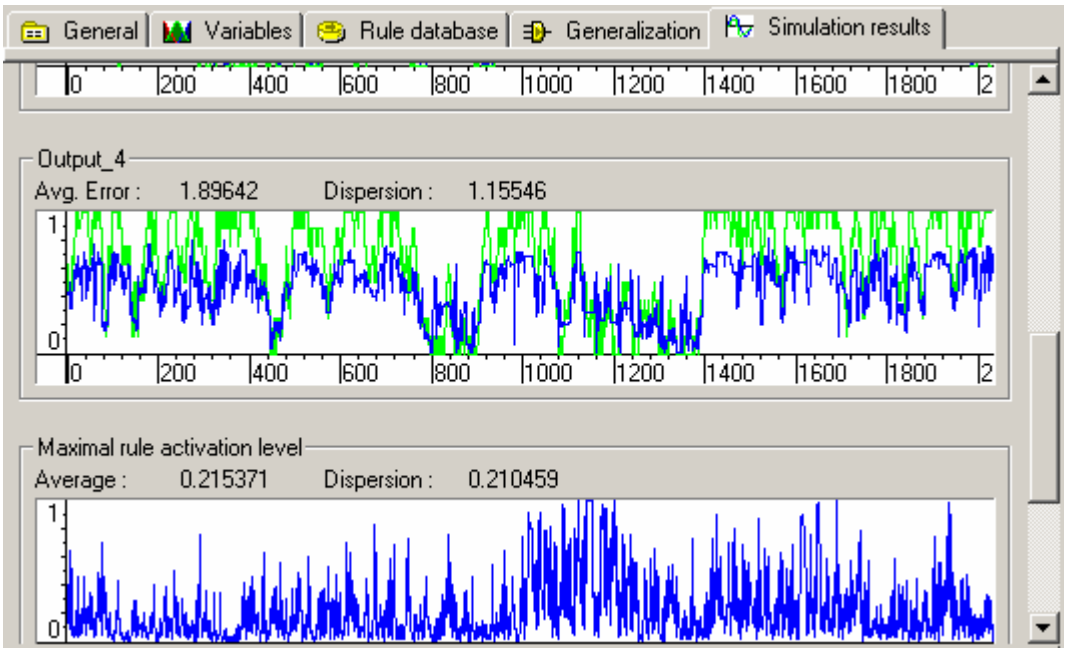

Figure 4.2.8.1. Simulation results page.

This page contains graphs with model outputs for all variables, and also graphs with maximal rule activation level and number of activated rules for each teaching signal line. It also displays numerical values with average error and error dispersion for all graphs. Use scroll bar on the right of the page to scroll page contents if it does not fit on single screen.

Each graph haves a pop-up menu that can be activated by right-clicking the mouse on the graph. This menu can be used to alter graph appearance and save image to file. Menu items are as follows:

`Display error interval`: Highlight regions for which output was not calculated properly.

`Display delta`: Display difference between teaching signal and model output instead of signals itself. In this mode green line will show 0-error level and blue line will show error

value. This mode is not available on `Number of activated rules` and `Rule activation level` graphs.
`Color lines`: Display signals with color (green and blue) lines.
`B&W lines`: Display signals with black lines.
`Save Image`: Save contents of the window as Windows BMP file.

4.2.9 Simulation Preview Page

Simulation preview page can be used to verify current model output and check effect of the changes without switching to `Simulation Result` page. Activate it by clicking the mouse on the `Simulation preview` tab or by selecting `View/Simulation preview` menu command.

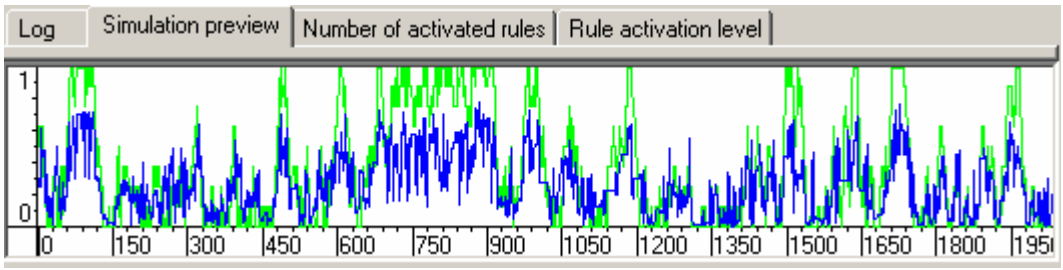

Figure 4.2.9.1. Simulation preview page.

`Simulation preview` window displays both **teaching signal** and **model output** for one of the output variables. The green line displays **teaching signal**. The blue line displays **model output**.

Regions highlighted with red background represent samples of **teaching signal** that do not have rules with corresponding if-part in the database. The model cannot calculate output for those samples.

`Simulation preview` window can be customized by the pop-up menu, activated by click of the right mouse button inside the window. Menu has following items:

`Variable`: Change output variable, which signals are displayed.
`Display error interval`: Highlight regions for which output was not calculated properly.
`Display delta`: Display difference between teaching signal and model output instead of signals itself. In this mode green line will show 0-error level and blue line will show error value.
`Color lines`: Display signals with color (green and blue) lines.
`B&W lines`: Display signals with black lines.
`Save Image`: Save contents of the window as Windows BMP file.

4.2.10 Number of activated rules Page

`Number of activated rules` preview page can be used to view number of activated rules graph without switching to `Simulation Result` page. Activate it by clicking the mouse on the `Number of activated rules` tab or by selecting `View/Number of activated rules` menu command.

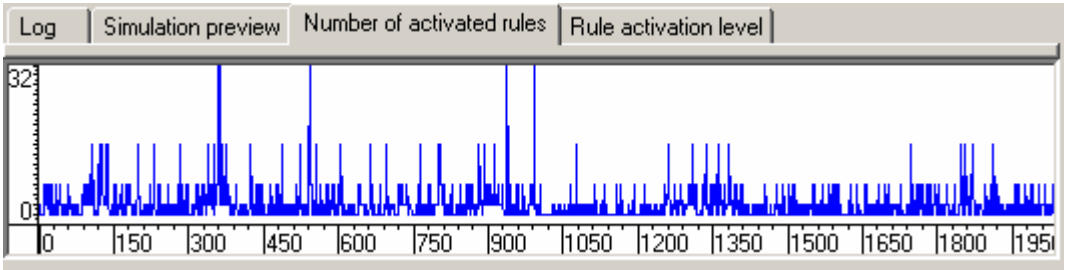

Figure 4.2.10.1 Number of activated rules page.

Regions highlighted with red background represent samples of **teaching signal** that do not have rules with corresponding if-part in the database. The model cannot calculate output for those samples.

Graph window can be customized by the pop-up menu, activated by click of the right mouse button inside the window. Menu has following items:

`Display error interval`: Highlight regions for which output was not calculated properly.
`Color lines`: Display signals with color (green and blue) lines.
`B&W lines`: Display signals with black lines.
`Save Image`: Save contents of the window as Windows BMP file.

4.2.11 Rule activation level Page

`Rule activation level` preview page can be used to view rule activation level graph without switching to `Simulation Result` page. Activate it by clicking the mouse on the `Rule activation level` tab or by selecting `View/Rule activation level` menu command.

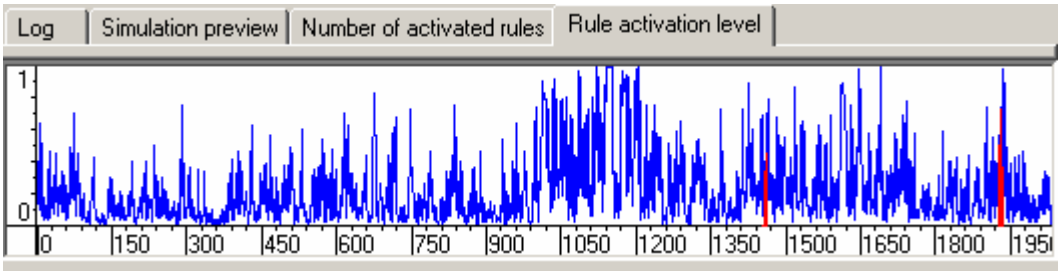

Figure 4.2.11.1 Number of activated rules page.

Regions highlighted with red background represent samples of **teaching signal** that do not have rules with corresponding if-part in the database. The model cannot calculate output for those samples.

This graph also displays activation level of rule selected in the `Rule database` page. Activation level of selected rule is displayed with red lines. This function is available only on the `Rule activation level` page, and not on the `Simulation Result` page.

Graph window can be customized by the pop-up menu, activated by click of the right mouse button inside the window. Menu has following items:

`Display error interval`: Highlight regions for which output was not calculated properly.
`Color lines`: Display signals with color (green and blue) lines.
`B&W lines`: Display signals with black lines.
`Save Image`: Save contents of the window as Windows BMP file.

4.2.12 Creating Variables

First task you should perform when creating the model is to create **membership functions** of the variables. Press `Create Variables` button or select `Action/Generate Variables` menu item to start MF creation.

SCOptimizer supports two modes of this operation. If you wish to select number of MF's per variable and their shape manually you should define MF shape using **uniform distribution algorithm**. Or, SCOptimizer may optimize those parameters using **GA1 algorithm**.

When you activate Create Variables function the wizard dialog will appear, which will first ask you about creation method. Select one and press `Next>>` to enter options.

4.2.12.1 Creating variables with GA-1 algorithm

When working with GA-1 algorithm you can run signal filtering algorithm which will remove redundant signal lines. This can improve quality of fuzzy sets created by GA-1 algorithm. If you wish

to use this mode select `Filter Signal` checkbox on the first page of the dialog and enter desired filter threshold level.

You can also check `Do not change number of MF` box if you wish GA-1 algorithm only alter shapes of MF's but not their number.

Next page will be the page with genetic algorithm parameters. Fill them and press `NEXT>>` to switch to the next page.

Select variables, which should be optimized, by holding `CONTROL` key and clicking items in the list. If you are running this algorithm for the first time it is recommended to leave all variables selected. Use this feature in order to improve quality of some variables later.

If you check `Add elements of the fitness vector` box then elements of the resulting fitness vector will be added together. Otherwise vector fitness function will be used. Press `NEXT>>` to start creation process.

While GA-1 algorithm operates the progress dialog will be shown. It will display number of current generation and achieved level of evaluation function. GA-1 uses different fitness functions for input and output variables, so it will first optimize input variables and then output.

You can press `Abort Stage` button if you want to stop optimization for the current stage. The state of the variables will be set to the best state found before abort button was pressed and the optimization will switch to the next stage. Press `Abort All` to stop optimization process and return to SCOptimizer.

4.2.12.2 Creating variables with Uniform distribution algorithm

For uniform distribution algorithm you should manually select shapes of membership functions. Wizard will display two identical pages with parameters for input and output variables:

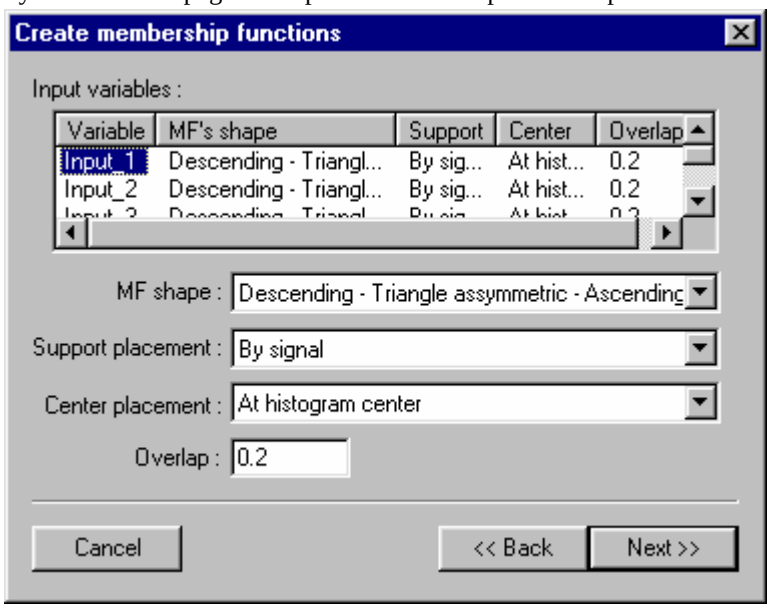

Figure 4.2.12.2.1. Uniform distribution algorithm parameters.

To select parameters for the variable select them from the list and change values in the fields below:

`MF Shape`: Shapes of the membership functions.

`Support placement`: Specify how to place supports. `By signal` means distribute supports so that each one will include equal number of **teaching signal** samples. `Uniformly` will distribute supports uniformly on the signal change interval.

`Center placement`: Specify how to place centers of non-symmetrical distributions. `At histogram center` will place center so, that there will be equal number of signal samples in the support area to the left and to the right from center. Closer to interval bounds will shift

centers of MF's to the nearest signal interval bound. Amount of shift increases with distance from interval center.

`Overlap`: Overlap coefficient between neighbor supports. Values from –1 to 1 are allowed, 0 meaning no overlap, positive values – overlapping supports, negative – space between supports.

When you have done with input variables press `NEXT>>` to switch to the next page. Fill parameters for output variables and press `NEXT>>` again. Uniform distribution algorithm will start and MF's will be created.

4.2.13 Creating Rule Database

After you have created variables and membership functions you can create **rule database**. You can do this by pressing `Create rule database` command button or with `Action/Create rule database` menu.

When you activate creation of rule database you will see following dialog box:

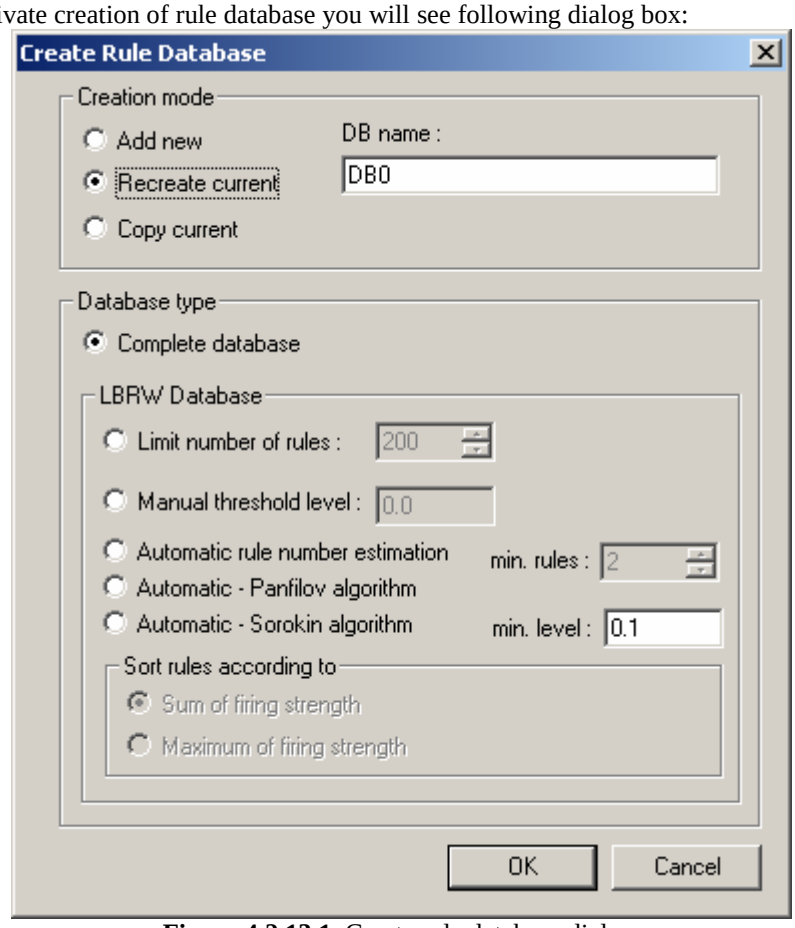

Figure 4.2.13.1. Create rule database dialog.

First select creation mode of the database. Following modes are available:

`Add new`: create new rule database and add it to the model. Old databases are not changed.

`Recreate current`: Replace current database with new one. Data in the active database is destroyed.

`Copy current`: Make a copy of current database. In this case creation mode settings are ignored.

You can also input a `name` of the database, or use an automatically generated one.

Then select type of rule database. **Complete rule database** will store all the rules for given model. Number of rules in complete database equals to product of numbers of fuzzy sets of input variables. This will result in extremely large database and very slow optimization speed if you will try to use it

with more than one-two input variables. If you wish to use this type of database, select `Complete database` and press `OK`. Database will be immediately created.

If size of complete database is not acceptable, select a **LBRW database**. LBRW database stores only selected number of rules, which are chosen by "**Let the Best Rule Win**" algorithm. When creating **LBRW database** you can specify exact number of rules or minimal level of firing strength (threshold level). In the latter case created database will include all rules with firing strength greater than or equal to one you specify.

You can also use automatic rule number estimation mode. In this case program will select minimal number of rules so, that the given number of rules will cover each point of teaching signal with level not less than minimal, if it is possible. For the moment there are three different automatic algorithms, which will produce different results.

You can also specify how **LBRW algorithm** should sort rules when selecting. If you select `Sum of firing strengths` then **LBRW algorithm** will add firing strengths of the rule for each sample of **teaching signal**. If you select `Maximum of firing strength` then only maximal value will be used.

If you wish to create **LBRW database** select desired creation mode, enter number of rules or threshold level as required and press `OK`.

LBRW algorithm requires some time to analyze a data and create a database.

4.2.14 Optimizing Rule Database

The database created by `Create Rule Database` command sets all outputs to 0. In order to fill you database with actual data you should optimize the database. SCOptimizer supports the only way to optimize rule database: **GA2** optimization. **GA2** uses **genetic algorithm** to optimize a database. Database states are analyzed by comparing inference output with reference signal and minimizing difference between those signals. Each output is optimized separately.

To start database optimization press `Optimize rules` command button or select `Action/Optimize rule database` menu item. `Rule Database Optimization` wizard will appear.

First select **teaching signal source** for **GA2** algorithm. `Complete` means using all data from currently selected **teaching signal** file. For large models this can result in slow optimization speed. If this is a case you can select `Optimized` signal.

If you select `Optimized` signal then **pattern reduction algorithm** will be used. This algorithm tries to leave only those samples of **teaching signal**, which activate different rules. This will increase **GA2** optimization speed without much lost in precision.

You can also use Matlab/Simulink simulation as source of data for **GA2** optimization. See section 4.2.17 for details on using this mode.

When you select signal source press `NEXT>>` to process to the page with **genetic algorithm** parameters. Change those parameters as required and press `NEXT>>` to start switch to the next page.

Now you should select output variables for which database should be optimized. By default optimization is selected for all variables and you shouldn't change it when starting algorithm for the first time. You can change selection by holding `CTRL` and clicking left mouse button on the list items. You also have an option to optimize all variables at the same time (if you check "`optimize all the variables at the same time`" check box). If you live this checkbox unchecked program will optimize variables one after another. If you check `Add elements of the fitness vector` box then elements of the resulting fitness vector will be added together. Otherwise vector fitness function will be used. Press `NEXT>>` to start optimization.

During optimization a progress window will appear. It will display which variable is currently optimized, number of current generation and achieved level of evaluation function.

You can press `Abort Stage` button if you want to stop optimization for the current stage. The state of the variables will be set to the best state found before abort button was pressed and the optimization will switch to the next variable. Press `Abort All` to stop optimization process and return to SCOptimizer.

After completing optimization you will see GA save state dialog.

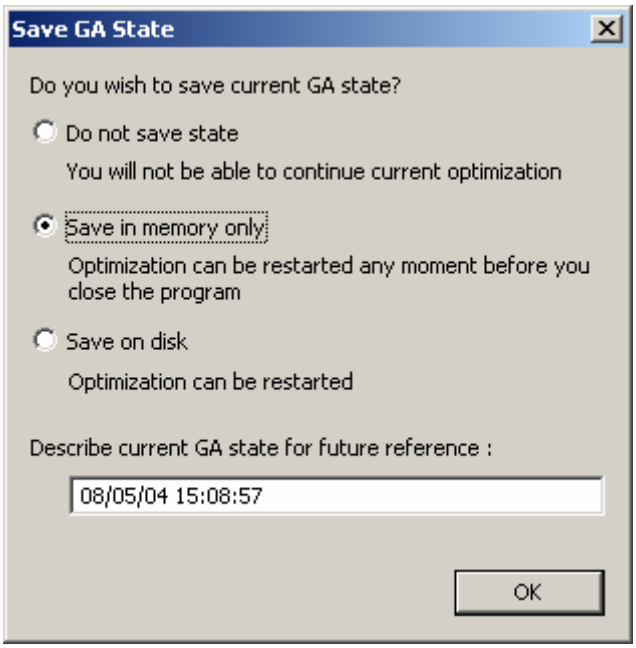

Figure 4.2.13.1. Save GA state dialog.

This dialog allows you to save genetic algorithm state, if you wish to restart this optimization process later. Three modes are available:

`Do not save`: state will not be saved and optimization can not be restarted.

`Save to memory`: state will be stored in program memory and will be available unless you close the program.

`Save on disk`: state will be saved in file on disk and optimization can be restarted even after you restart the program.

You can also enter state description, which will be displayed when you select state to continue optimization. By default state description contains date and time when optimization was stopped.

To use previously saved state you should check `Use previous state as initial` checkbox and select state description in the combo box on the GA options page in the GA wizard dialog.

4.2.15 Model Refinement

When rule database is optimized you can further improve model quality by returning to **MF** optimization. This is accomplished by the last optimization step – model refinement (known as **GA-3** algorithm). You can start model refinement by clicking `Refine KB` command button or selecting `Action/Refine KB` menu item.

After you activate the command wizard dialog will appear. It will first prompt you which fitness function you would like to use. Three choices are available:

`Maximization of mutual information entropy`: Tells SCOptimizer to minimize mutual information entropy between MF fuzzy sets. This is the same function used

in GA-1 algorithm, but unlike GA-1, GA-3 won't change number of MF's per variable, only MF parameters will be changed.
`Minimization of output error`: Minimize output error.
`Matlab simulation`: Use Matlab/Simulink to calculate fitness function. See section 4.2.17 for details on using Matlab interface.

Select one of the variants and press `NEXT>>`. Enter genetic algorithm parameters on the second page and press `NEXT>>` to switch to the next page.

Now you should select input variables, which should be optimized. By default optimization is selected for all variables. You can change selection by holding `CTRL` and clicking left mouse button on the list items. You also have an option to optimize all variables at the same time (if you check "`optimize all the variables at the same time`" check box). If you leave this checkbox unchecked program will optimize variables one after another. If you check `Add elements of the fitness vector` box then elements of the resulting fitness vector will be added together. Otherwise vector fitness function will be used. Press `NEXT>>` to start optimization.

While **GA-3** algorithm operates the progress dialog will be shown. It will display number of current generation and achieved level of evaluation function.

You can press `Abort Stage` button if you want to stop optimization for the current stage. The state of the variables will be set to the best state found before abort button was pressed and the optimization will switch to the next variable. Press `Abort All` to stop optimization process and return to SCOptimizer.

If you are still not satisfied with model quality you can run rule database optimization (**GA-2**) again or use **Error Back Propagation** algorithm.

4.2.16 Using Error Back Propagation algorithm

Error Back Propagation algorithm implements classical gradient optimization method, which provides an effective way to further improve model output after genetic optimization. You can start Back Propagation algorithm by clicking `Back Propagation` command button or selecting `Action/Back Propagation` menu item.

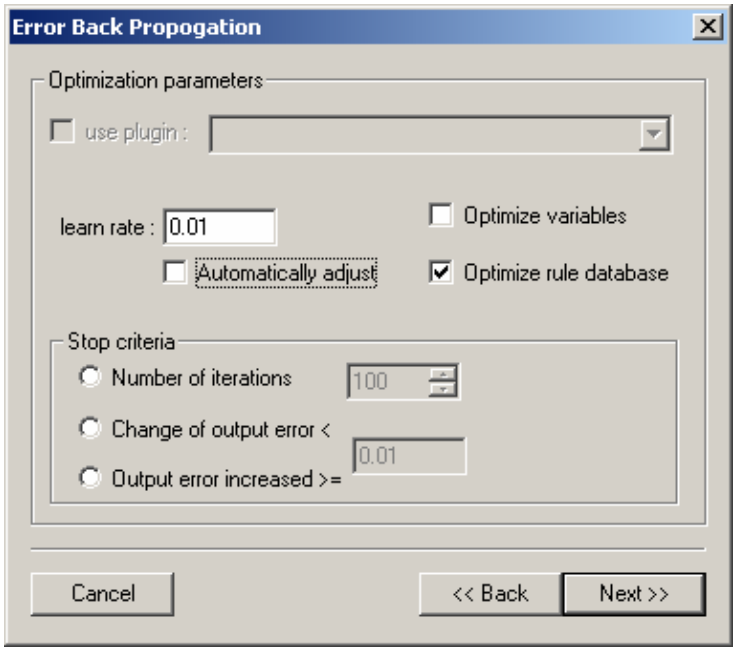

Figure 4.2.16.1. Back propagation dialog

Back propagation algorithm can use **teaching signal** or use plugin module for optimization. If you want to use plugin, check use plugin checkbox and select plugin name. Those controls are only available if at least one plugin that supports back propagation mode is installed.

After you activate back propagation algorithm a wizard dialog will appear. On the first page you should enter algorithm parameters: `learn rate`, which define how much model parameters should be changed in response to the output error, and stop criteria: algorithm can be stopped after fixed number of iterations or when change of output error became less then given threshold. `Automatically adjust` check box allow you to enable automatic lean rate adjustment mode. Fill this parameters and press `NEXT>>` to switch to the next page.

Now you should select input variables, which should be optimized. By default optimization is selected for all variables. You can change selection by holding `CTRL` and clicking left mouse button on the list items. Press `NEXT>>` to start optimization.

You can press `Abort Stage` button if you want to stop optimization for the current stage. The state of the variables will be set to the best state found before abort button was pressed and the optimization will switch to the next variable. Press `Abort All` to stop optimization process and return to SCOptimizer.

4.2.17 Creating databases with incomplete rules

SCOptimizer has an option for creating rule databases with incomplete rules. In order to create such a database you should first create and optimize normal database.

To create incomplete database press `Incomplete rules` button. Following dialog will appear:

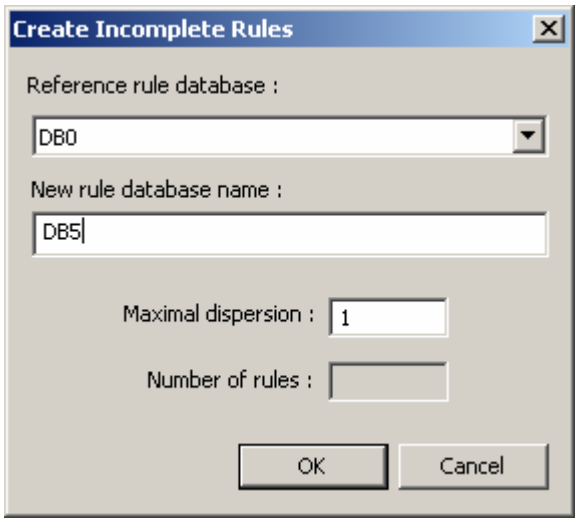

Figure 4.2.17.1. Incomplete rules creation dialog

First select database, which will be used as reference while creating DB with incomplete rules. Incomplete rules creation algorithm analyses reference rule database and finds rules, which can be joined together to form incomplete rules. Rules are joined if dispersion of their outputs is less than a value entered in `Maximal dispersion` field.

Enter name of the new database and press `OK` to create it.

4.2.18 Using Matlab/Simulink for model optimization

You can use Matlab/Simulink to calculate **fitness function** for genetic algorithms during creation of the model.

SCOptimizer uses following model for interaction with Matlab. When genetic algorithm **require fitness value** for current model state it executes Matlab function specified by the user. This function in

turn should make calls to SCOptimizer library functions GAInfer/SimGAInfer that will perform inference operation with current model. Matlab function should compute **fitness function** based on the model output and return it to SCOptimizer.

When you wish to use Matlab for calculation you should first select Matlab as a source of **fitness function**. If you do new page will appear in the wizard after all pages. This will be the page with following parameters:

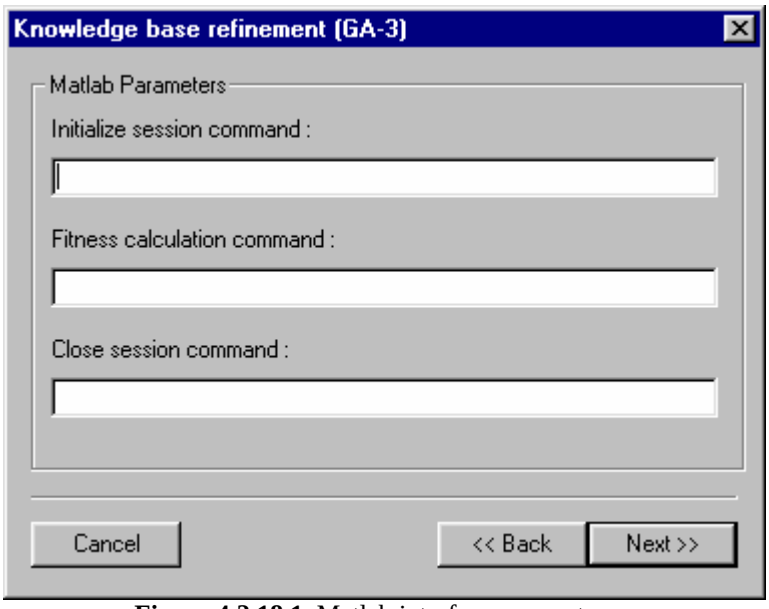

Figure 4.2.18.1. Matlab interface parameters page.

You should enter three Matlab commands on this page:

`Initialize session command`: This command will be executed at the beginning of the genetic optimization step. It will have one string parameter, which should be passed to GAConnect/SimGAConnect function to initialize session with SCOptimizer.

`Fitness calculation command`: This command will be called whenever fitness value is required. This command should calculate **fitness value** using GAInfer/SimGAInfer and place it to variable called SCO_Fitness.

`Close session command`: This command is called when genetic optimization is over. It is expected to call GADisconnect/SimGADisconnect to close Matlab-SCOptimizer link and free other resources, if any. If you do not specify close session command then Matlab session won't be closed when optimization is over. You can use this if you wish to have access to Matlab after optimization, for example to review intermediate parameters or to debug fitness function.

When genetic optimization starts SCOptimizer will start Matlab session and execute above commands as Matlab command. Parameter will be added to initialize command by adding "('xxxx')" to the text specified in dialog box (where 'xxxx' will be replaced by some automatically-generated string).

See section 6.2: SCLib interface for Matlab fitness function calculation for the description of functions, which should be used in Matlab environment to perform fuzzy inference.

4.2.19 GA Test mode

GA Test mode can be used to perform optimization of abstract variables using Matlab for fitness value calculations. For example this mode can be used to generate teaching signal for SCOptimizer. You can start GA Test algorithm by clicking `GA Test` command button or selecting `Action/GA Test` menu item.

On the first page of GA Test wizard enter genetic algorithm parameters and press `NEXT>>` to switch to the next page.

Now you should define parameters, which should be optimized. Parameters are entered in groups, parameters inside the group has equal characteristics. For each group you should enter number of elements in group, minimal and maximal values and search step (actual step can be less than specified, based on number of bits used to represent a value). List at the top part of the window displays currently defined groups of parameters. Line below the list graphically displays parts of the chromosome used to represent each group. Buttons and entry fields below the list are used to add, change and delete groups. `Optimize groups together` checkbox is used to tell SCOptimizer to optimize all groups at the same time. If it is not checked groups will be optimized separately.

When you are over with parameters press `NEXT>>` to switch to the next page. Now you see standard Matlab commands page. It works the way described in section 4.2.17, expect parameter pass agreement. SCOptimizer adds following text to the fitness calculation command : (N,[x1,…,xl]), where N=0 for “optimize groups together” mode, index of the group otherwise. X1,…,Xl - values of the currently optimized parameters (current group or all groups).

Press `Next>>` to start optimization.

You can press `Abort Stage` button if you want to stop optimization for the current stage. The state of the variables will be set to the best state found before abort button was pressed and the optimization will switch to the next group. Press `Abort All` to stop optimization process and return to SCOptimizer.

5 Using Inference

Inference is a stand-alone tool designed to simulate a fuzzy system behavior in order to verify the approximation level obtained during learning phase or to use inference process from other applications.
Inference can work in two modes: simulation on a single pattern or simulation from file.
Inference is a command-line program. You should run it from MS-DOS window or with Start->Run Windows menu.

5.1.1 Simulation on a single pattern

If you wish to get a fuzzy system output for a single pattern run Inference with the following arguments:

```
inference.exe model.sco 0.5 1.2
```

Where **`model.sco`** is a name of file of model, created by SCOptimizer. **`0.5`** and **`1.2`** are input values for the first and for the second input variable correspondingly. You should specify input values for all input variables of model.
If all parameters are set correctly, program will respond in following way:

```
Input_1=0.5 Input_2=1.2 Output_1=0.3 Output_2=0.9
```

Here `Input_1` and `Input_2` are names of input variables, `0.5` and `1.2` are their values, as specified in command line. `Output_1` and `Output_2` are names of output variables, and `0.3` and `0.9` are calculated output values.

5.1.2 Simulation from file

To simulate a fuzzy system on a large number of input vectors you can use simulation from file mode. To use Inference in this mode start it with following command:

```
inference.exe model.sco in_file.[txt|mat] out_file.txt
```

Where **`model.sco`** is a name of the file with model created by SCOptimizer. **`In_file`** and **`out_file`** are names of the input file and of the output file correspondingly.
Inference can read both Matlab and text files for input. This file should have the same format as learning signal, i.e. it should have columns of data for input and output variables. If you use text files their format should match text format specified for your model. If this is not a case load a model with SCOptimizer and change number format settings.
Program will write calculated output values for each input pattern to output file. Only text mode output files are supported. Numbers are written in format, saved in model file. Outputs for different input vectors are separated with new line characters.

6 Using stand alone inference library from C++ code

You can use models, created with SCOptimizer to perform inference calculations from C++ code, using SCLib library module supplied with SCOptimizer.
For simple inference operations this module defines 3 functions:

- BOOL SCLoad(const TCHAR * name,InferenceEngine **Engine) – Loads model from file. Parameter name is a pointer to a string containing SCOptimizer model. Function returns TRUE if operation was successful. FALSE is returned in case of error.
- FloatVector SCInfer(FloatVector & in,InferenceEngine *Engine,BOOL all) – Performs fuzzy inference. In is an input data vector. If all is set to true, inference is performed on all databases and generalized, if all is false only active database is used. Function returns vector, containing inference result. In case of error empty vector is returned.
- void SCFree(InferenceEngine **Engine) – Frees memory allocated by SCLoad().

Class FloatVector used as input and output for inference procedure is a dynamic array of floating point numbers. Class interface includes following functions:

- int GetSize() – returns current size (number of elements) of array.
- void SetSize(int newSize) – sets size of array to newSize elements.
- float & ElementAt(int Index) – returns reference to element at position Index.
- float & operator [](int Index) – same as ElementAt().
- void SetAt(int Index,float newEl) – sets element Index of array to newEl. Element must exist in array, i.e. Index must be less then value, returned by GetSize().
- void SetAtGrow(int Index,float newEl) - sets element Index of array to newEl. Array size is automatically increased if Index is greater then array size.
- void Add(float newEl) – Add newEl to the end of array.
- void RemoveAll() – delete all elements and set size of array to 0.
- void Mark(size_t Index,BOOL Exist) – set exist flag for variable at Index.
- void MarkAll(BOOL Exist) – set exist flag for all variables.
- void Remove(size_t Index) – set exist flag of variable Index to false.
- BOOL Exist(size_t Index) – return exist flag for variable Index.

You can also use other classes and functions of SCOptimizer, which are described in Appendix 2 – SCLib Class hierarchy.

Here is an example of a program using SCLib library module:

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#include "SCLib\sclib.h"

InferenceEngine *Engine;

int main(void)
{
 TCHAR name[_MAX_PATH];
 FloatVector in,out;
 int i;


 // Get model file name and load model
 _tprintf(_T("Enter model file name : "));
 _getts(name);

 // loads engine from file
 if(!SCLoad(name,&Engine))
 {
       _tprintf(_T("Error loading file\r\n"));   // load failed
```

```
        return 1;
    }


    // set size of input vector
    in.SetSize(Engine->GetInputVarCount());

    // query input data
    for(i=0;i<Engine->GetInputVarCount();i++)
    {
        _tprintf(_T("Enter value for variable %s: ")
                    ,(LPCSTR)Engine->InputVar(i).name);
        scanf("%f",&(in[i]));
    }

    // infer
    out=SCInfer(in,Engine,true);

    // check if inference returned data
    if(out.GetSize()<Engine->GetOutputVarCount())
    {
        _tprintf(_T("Inference process failed!\r\n"));
        return 2;
    }

    // printf result
    _tprintf(_T("\r\n"));
    for(i=0;i<out.GetSize();i++)
    {
        _tprintf(_T("%s: %g\r\n"),
                    (LPCSTR)Engine->OutputVar(i).name,out[i]);
    }

    // free memory and exit program
    SCFree(&Engine);

    return 0;
}
```

6.1 Using SCLib library from Simulink

SCLib library includes functions, which support Simulink workspace interface for storing engine pointer. This functions are the following:

- BOOL SimSCLoad(const TCHAR *name, SimStruct * S);
- FloatVector & SimSCInfer(FloatVector &in, SimStruct *S, bool all);
- void SimSCFree(SimStruct *S);

They have the same functionality as previously described functions without Sim prefix but use SimStruct to store engine pointer between calls.

You can use the following code to get pointer to InferenceEngine object, if you wish to use some other functions:

```
void **pWork = ssGetPWork(S);
InferenceEngine *Engine = (InferenceEngine*)pWork[0];
```

6.2 SCLib interface for Matlab fitness function calculation

When you use Matlab as source of teaching data for genetic algorithms in SCOptimizer you should use special functions to perform fuzzy inference.

- BOOL GAConnect(LPCTSTR param) – called to establish connection with SCOptimizer process. Param is a text string passed by SCOptimizer as parameter of Initialize Session command. This function should be called before first call to GAInfer().
- FloatVector & GAInfer(FloatVector &in) – performs inference using current SCOptimizer state.
- void GADisconnect() – frees resources associated with SCOptimizer link. You should call this function once for each call of GAConnect().

Other calls are not supported in this mode.

Simulink versions of those functions are also available:

- BOOL SimGAConnect(LPCTSTR param, SimStruct *S);
- FloatVector & SimGAInfer(FloatVector &in, SimStruct *S, bool all);
- void SimGADisconnect(SimStruct *S);

7 Appendix 1 – Supported MF distributions

SCOptimizer supports following fuzzy membership function shapes:

- Exact numbers
  MF of exact numbers equals 1 at some value and 0 in all other cases.
  Exact numbers are written like they are: 1 (exact number “one”).
- Triangular
  MF of triangular distribution equals to 1 at modal_value and linearly decreases to 0 at modal_value – left_fuzzy and modal_value + right_fuzzy points.
  Triangular distribution is written in the following form:
  tr(modal_value;left_fuzzy;right_fuzzy). Example: tr(0;1.5;1.3).
- Trapezium
  MF of trapezium distribution equals 1 at interval [left_tolerant,right_tolerant] and linearly decreases to 0 at left_tolerant-left_fuzzy and right_tolerant+right_fuzzy points.
  Trapezium distribution is written as tp(left_tolerant;right_tolerant;left_fuzzy;right_fuzzy). Example: tp(0;1;0.5;0.5).
- Descending
  MF of descending distribution equals to 1 at all points less then or equal to modal_value, then it linearly decrease to 0 at modal_value+fuzzy point and remains 0 for greater values.
  Descending distribution format is: ds(modal_value;fuzzy). Example: ds(1;0.5).
- Ascending
  MF of ascending distribution equals 0 at points less than modal_value-fuzzy, then it linearly increase and reach 1 at modal_value. At points greater than modal_value it equals 1.
  Ascending distribution format is: as(modal_value;fuzzy). Example: as(0;0.3).
- Normal (Gaussian)
  MF of normal distribution is changed according to Gaussian function: $\exp(-9*(x-\text{modal\_value})^2/(2*\text{fuzzy}^2)$
  Normal distribution format is n(modal_value;fuzzy). Example: n(0;1).
- Asymmetrical normal
  Asymmetrical normal distribution has a shape of Gaussian function with different scale parameters to the left and to the right from modal value.
  Format for asymmetrical normal distribution is: asymn(modal_value;left;right).
  Example: asymn(0;1.1;0.9).
- Normal descending
  MF of descending normal distribution equals to 1 at all points less than or equal to modal_value, then it have shape of gaussian function.
  Descending normal distribution format is: descn(modal_value;fuzzy). Example: descn(1;0.5).
- Normal ascending
  MF of ascending normal distribution equals to 1 at all points greater than or equal to modal_value, at lower values it have a shape of Gaussian function.
  Ascending normal distribution format is: ascn(modal_value;fuzzy). Example: ascn(1;0.5).

Please note, that numbers should be written in the format, specified for model. Examples are given for the case when decimal separator is dot (.) and list separator is semicolon (;). See **changing locale settings** section for more details.

8 Appendix 2 – SCLib Class hierarchy

8.1 Main ideas

The class structure of the SCOptimizer library is presented on the Figure A2.1.
Base of the system is composed of classes, implemented fuzzy inference algorithm (child of InferenceEngine class). Each of these objects encapsulates arrays of LinguisticVariable objects, representing input and output variables. Each LinguisticVariable object encapsulates array of FMbF objects – fuzzy membership functions (Figure A2.2.)

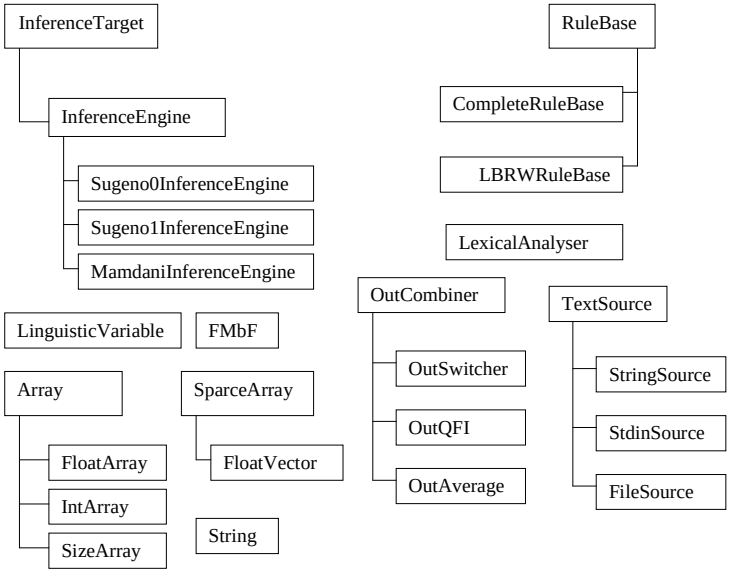

Figure A2.1: Class hierarchy of Soft Computing Optimizer

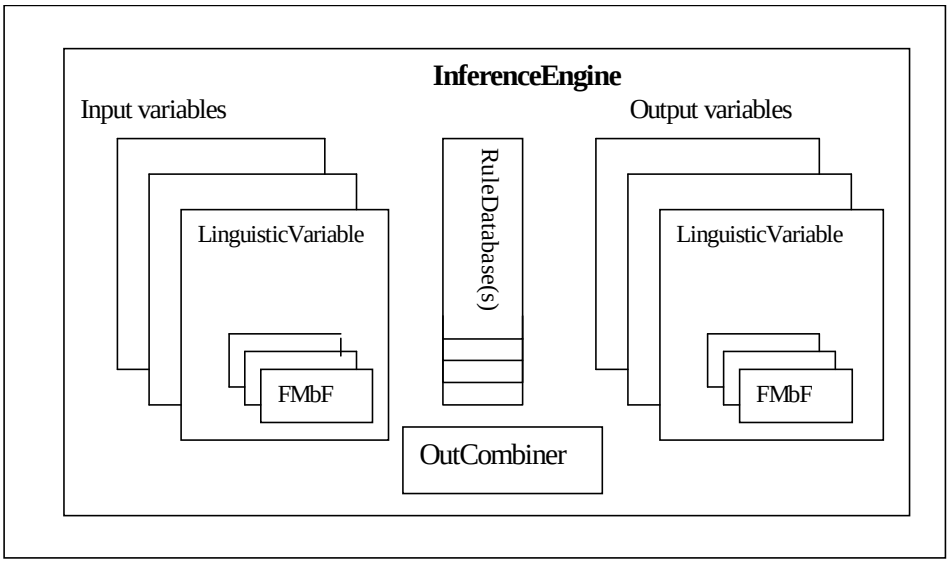

Figure A2.2: Objects used during fuzzy inference procedure.

Fuzzy inference rules are stored with the help of RuleBase class and it's child classes. Currently two types of rule databases are supported (in corresponding classes): complete rule database stores complete set of rules and LBRW database stores only rules, selected with the help of LBRW algorithm (only rules with higher firing strength are stored). RuleBase objects only provide storage and access to rule data, they do not deal with it internal structure.

Fuzzy inference algorithm, implemented in InferenceEngine class, detects rules, which are active for given input and sends their numbers to a InferenceTarget class. InferenceTarget can be a InferenceEngine or user-supplied class. InferenceEngine extracts rules from database and perform required calculations.

OutCombiner and derived classes are responsible for output generalization, i.e. they process output vectors resulting from inference with different rule databases and calculate single resulting vector. OutAverage and OutQFI represent weighted average and quantum generalization algorithms. OutSwitcher class is used to switch between those algorithms.

Lexical analyzer, implemented in LexAnalyser class, is used to read model data, text data files and process user input. It converts stream of characters to stream of lexemes, representing words, numbers and different separators. TextSource class and its child classes represent input stream for LexAnalyser. They allow LexAnalyser to be used to process files, memory strings and direct user input.

Supplementary classes, representing dynamic arrays, text strings, and others are used by other classes.

8.2 Class description

8.2.1 FMbF Class

– represents single fuzzy membership function.

8.2.1.1 Class Variables:

- FDistr _distribution – distribution type. FDistr is enumerated type, following values are allowed: exact, triangular, trapezium, ascending, descending, normal,asym_normal,asc_normal,desc_normal.
- float _modalvalue - modal value. Used with all distribution types except trapezium.
- float _righttolerant, _lefttolerant – right and left margins of tolerant interval, used with trapezium distribution.
- float _rightfuzzy – right fuzzy parameter. Used with normal, asym_normal,triangular, trapezium, descending and desc_normal distribution types.
- float _leftfuzzy – left fuzzy parameter. Used with triangular, trapezium, asym_normal, asc_normal and ascending distribution types.
- String _name – human-given distribution name.

8.2.1.2 Constructors:

- FMbF() – creates object representing “exact 0”.
- FMbF(float f,string name) – creates object representing “exact f”.
- FMBF(FMbF &a,string name) – creates object with distribution taken from object a.

8.2.1.3 Operations:

- float Fitness(float x) – returns fitness value at point x.
- float GetAMinus(float a) – returns left margin of a-level (minimal x such that Fitness(x)>=a).
- float GetAPlus(float a) – returns right margin of a-level (maximal x such that Fitness(x)>=a).
- BOOL Load(LexAnalyser &l) – read object state from LexAnalyser. Input string have the following format:
  -exact: _modalvalue
  -triangular: tr(_modalvalue,_leftfuzzy,_rightfuzzy)
  -trapezium: tp(_lefttolerant,_righttolerant,_leftfuzzy,_rightfuzzy)
  -ascending: as(_modalvalue,_leftfuzzy)
  -descending: ds(_modalvalue,_rightfuzzy)
  -normal: n(_modalvalue,_rightfuzzy)
  -asym_normal: asymn(_modalvalue,_leftfuzzy,_rightfuzzy)

-asc_normal: ascn(_modalvalue,_leftfuzzzy)
-desc_normal: descn(_modalvalue,_rightfuzzy)
_name is not changed during load. Returns TRUE on success, FALSE if error occurred.

- FMBF & operator = (FMBF &a) – copy object a to current object.
- float SupportMinus() – returns left margin of support (for normal distributions returns margin of interval where fitness value is not too small). If support is not limited then – FLT_MAX is returned. Value is returned in normalized space.
- float SupportPlus() – returns right margin of support (for normal distributions returns margin of interval where fitness value is not too small). If support is not limited then FLT_MAX is returned. Value is returned in normalized space.
- void SetModal(float min, float max) – sets tolerant interval margins or modal value. For trapezium distribution tolerant interval is set to [min,max], for ascending and descending normal distributions modal value is set to min, for all other distributions modal value is set to max. Min and Max values should be in normalized form.
- void SetSupport(float min,float max) – sets support interval to [min,max]. Left and right fuzzy parameters of distribution are changed to cover [min,max] interval. Min and Max values should be in normalized form.
- BOOL Unimodal(void) – returns TRUE if distribution is unimodal.

8.2.2 LinguisticVariable Class

- input/output linguistic variable.

8.2.2.1 Class Variables:

- static String name_base – Base name for FMbF name generation. When the LinguisticVariable object is created, enclosed FMbF's are named with names composed of name_base and integer index name_idx. Code which create LinguisticVariable object should set name_base and name_idx before creating the object. LinguisticVariable class never change value of this variable.
- static int name_idx – Integer index used for generation of FMbF's names. It is incremented by LinguisticVariable class constructor each time it assigns a new name for FMbF.
- String name – human-given variable name.
- FMbF *fun – pointer to array of FMbF's representing possible membership functions for given variable.
- int fun_num – number of FMbF's in fun array.
- float *fit – fitness values of each membership function for given input value. It is filled by CalcFitness function and used to speedup inference procedure.
- float offset, scale – parameters of normalization. Normalized value=input*scale+offset.
- float min,max – minimum and maximum values of signals.
- int *_act_idx – indexes of activated (fit>0) functions.
- int _act_num – number of elements in _act_idx.

8.2.2.2 Constructor:

- LinguisticVariable() – creates empty object. Constructor creates DEFAULT_FMBF_NUM membership functions (all equal exact 0), and gives them names composed of name_base and name_index. If constructor operation was unsuccessful (due to lack of memory), number of membership functions is set to 0.

8.2.2.3 Operations:

- int NumF() – returns number of membership functions
- FMbF & operator [] (int i) – allows access to i-th membership function. If index is out of range internal static variable is accessed.
- BOOL SetNumF(int num) – set number of membership functions. If num is greater than old count of MF's then old MF's are saved and new MF's are added. The distributions of

new MF's are set to exact 0 and names are generated from name_base & name_idx. If new count is less than old one, then MF array is truncated and parameters of first num FMbF's are saved. Code calling this function is responsible to call InferenceEngine::ResizeRulebase() so that InferenceEngine can take account of MF's number change.

- BOOL CalcFitness(float x) – Calculates fitness values for all membership functions and fills fit array with them. FALSE is returned to indicate error (fun or fit arrays are not allocated).
- BOOL FromMargins(float *margin,float *center,int mode,float overlap) – set MF's distributions so, that they will cover centroids, set by margin array. Array must have NumF()-1 element, representing margins between centroids. Centers of centroids are given in *center array, which must have NumF() elements. Mode variable selects shape of created MF's, according to following flags:
  - FM_FLAT_EDGES: if set, first and last MF will be of descending/ascending style.
  - FM_ASYM: asymmetrical distributions will be used if set, symmetrical if not.
  - FM_NORMAL: normal distributions will be used if flag is set, triangular otherwise.

  Overlap sets overlap coefficient between neighbor MF's (1 means no overlap, >1 – overlap, <1 – spacing between MF). Data and Margins arrays data is given in non-normalized form.
- float Normalize(float f) – normalize input value. Returns f*scale+offset.
- float DeNormalize(float f) – denormalize value. Returns (f-offset)/scale.
- void SetNormalization(float _min,float _max, float floor,float ceil, float margin) – Sets normalization parameters so, that input interval [_min-m,_max+m] maps to interval [floor,ceil], where m=(_max-_min)/margin.
- BOOL Load(LexAnalyser &l) – restore object state from LexAnalyser.
- int MaxSupport(float f) – returns index of rightmost support which covers f (f should be given in normalized form).
- int MinSupport(float f) – returns index of leftmost support which covers f (f should be given in normalized form).

8.2.3 InferenceTarget Class

- Represents target for fuzzy inference algorithm.

8.2.3.1 Operations:

- BOOL ActivateRule(int *index, FloatVector & v) – *implemented in child classes* - called during inference procedure by inference algorithm when it finds that a rule identified by array index (each element of index array is a index of FMbF of corresponding input variable included in the left part of the rule), is active for input vector v. InferenceTarget is expected to carry out all necessary calculations inside this function. Returning FALSE it signals that something goes wrong and inference procedure should be terminated.

8.2.4 InferenceEngine Class

- Base class for different fuzzy inference methods. Implementation of specific inference methods will be placed in child classes, like SugenoInferenceEngine, MamdamiInferenceEngine, etc.

8.2.4.1 BaseClass: InferenceTarget

8.2.4.2 Class Variables:

- UINT State – current model state. Supported values are:
  - ES_NOTHING – engine not initialized
  - ES_NORMALIZED – normalization coefficients are set
  - ES_MF_CONSTRUCTED – MFs are constructed
  - ES_BASE_CREATED – rule database created
  - ES_BASE_OPTIMIZED – rule database optimized
- int num_InputVars – number of input variables.

- LinguisticVariable * InputVars – pointer to array of input variables.
- int num_OutputVars – number of output variables.
- LinguisticVariable * OutputVars – pointer to array of output variables.
- RuleBase * rules – pointer to RuleBase (rule database storage class).
- BOOL MinInferenceMode – if true min() is used as fuzzy AND, if false product is used.
- BOOL ClippingDisabled – if FALSE output of inference procedure will be clipped to the output variable's signal change interval.

8.2.4.3 Constructor:

- InferenceEngine() – create object with 0 input/output variables and CompleteRuleBase.

8.2.4.4 Operations:

- BOOL UseOutputMF() – *implemented in child classes* – returns TRUE if MFs of output functions are used for inference calculations (Mamdani model), FALSE if not (Sugeno). InferenceEngine version returns 0.
- void SetInputVarCount(int i) – set number of input variables. If number of input variables and number of output variables is greater than 0 then ResizeRulebase is called.
- int GetInputVarCount(void) – get number of input variables.
- void SetOutputVarCount(int i) – set number of output variables. If number of input variables and number of output variables is greater than 0 then ResizeRulebase is called.
- int GetOutputVarCount(int i) – get number of output variables.
- LinguisticVariable & InputVar(int i) – access i-th input variable.
- LinguisticVariable & OutputVar(int i) – access i-th output variable.
- ResizeRulebase() – *implemented in child classes* – resize rule database according to change of number of variables or functions.
- BOOL SetInferenceMode(BOOL MinMode) – sets operation used for fuzzy AND. TRUE selects min(), FALSE selects product. Returns TRUE if required mode is supported.
- FloatVector InferAll(FloatVector InData) – Start fuzzy inference procedure on all databases and generalize result. On error empty (number of elements =0) vector is returned.
- FloatVector Infer(FloatVector InData) – Start fuzzy inference procedure on single database, make calculations for given InData vector and return result. On error empty (number of elements =0) vector is returned. The function calls PrepareInfer(), DoInfer() with target set to current object and FinalizeInfer() functions.
- BOOL DoInfer(FloatVector & InData,InferenceTarget &tgt) – perform fuzzy inference algorithm. Function find all rules, that are active for input vector InData and calls InferenceTarget's ActivateRule function for all found rules. If it is used for fuzzy inference procedure with InferenceEngine-based class then one should call PrepareInfer() before calling DoInfer() and FinalizeInfer() to get a result.
- BOOL PrepareInfer() – *implemented in child classes* – prepare object for fuzzy inference procedure. Object should allocate memory and initialize all variables required for fuzzy inference procedure. Called in the beginning of inference procedure.
- FloatVector & FinalizeInfer() – *implemented in child classes* – do all final calculations (if necessary) and return result of fuzzy inference. Called at the end of inference procedure.
- BOOL ActivateRule(int *index,FloatVector &v) – *implemented in child classes* –called during inference procedure by inference algorithm when it finds, that a rule identifyed by index, is active for input vector v.
- BOOL VarsByUniform(SignalSource &s, FloatVector overlap, IntArray mode, int num_Vars) – calculate input variable's MF's according to adaptation to uniform distribution method. This function creates SortedSource object from given source s and calls VarsByUniform(SortedSource …).
- BOOL Load(LexAnalyser &l) – restores object state from LexicalAnalyser. InferenceEngine implementation loads numbers of input and output variables, input variables and rule database. Child classes may implement their own version if required.

- BOOL LoadRule(int *index,LexAnalyser &l) – *implemented in child classes* – restores state of single rule from Lexical analyzer input. Returns TRUE on sucess, FALSE on error.
- void *AllocateRules(size_t num) – *implemented in child classes* – allocate memory for num rules and initialize it.
- size_t RuleLen(void) – *implemented in child classes* – return number of bytes required to store single rule.
- static InferenceEngine * GetEngineByCode(WORD code) – create child class object based by code. Currently 3 codes are supported: EC_SUGENO_0 – Sugeno0InferenceEngine, EC_SUGENO_1 – Sugeno1InferenceEngine & EC_MAMDANI - MamdaniInferenceEngine.
- static InferenceEngine * GetEngineFromFile(LexicalAnalyser &l) – parse data file header, determine engine type, create engine and call it's Load() function
- int GetEngineCode(void) – *implemented in child classes* – returns engine code (currently EC_SUGENO_0, EC_SUGENO_1 or EC_MAMDANI). Base class will return 0 to indicate error.
- String DescribeOutput(int *idx,int output) – *implemented in child classes* – returns TRUE if MFs of output functions are used for inference calculations (Mamdani model), FALSE if not (Sugeno). InferenceEngine version returns 0.
- BOOL DeleteMF(int var,int mf) – *implemented in child classes* – returns TRUE if MFs of output functions are used for inference calculations (Mamdani model), FALSE if not (Sugeno). InferenceEngine version returns 0.
- void DeleteBase(void) – deletes current rule database.
- int GetBaseCount(void) – returns number of rule databases.
- void SelectBase(int index) – set active database.

8.2.5 Sugeno0InferenceEngine Class

- Sugeno order 0 inference model implementation. Implemented according to InferenceEngine specifications.

8.2.5.1 BaseClass: InferenceEngine

8.2.6 Sugeno1InferenceEngine Class

- Sugeno order 1 inference model implementation. Implemented according to InferenceEngine specifications.

8.2.6.1 BaseClass: InferenceEngine

8.2.7 MamdaniInferenceEngine Class

- Mamdani inference model implementation. Implemented according to InferenceEngine specifications.

8.2.7.1 BaseClass: InferenceEngine

8.2.8 RuleBase Class

- Base class for rule database storage.

8.2.8.1 Class Variables:

- InferenceEngine &_engine – engine for wich to store data.
- size_t _rule_len – number of bytes required to store single rule.
- int _type – rule database type. 1 for CompleteDatabase, 2 for LBRWRuleDatabase.

8.2.8.2 Constructor:

- RuleBase(InferenceEngine &e) – sets _engine = e and _rule_len=0.

8.2.8.3 Operations:

- int Resize() – *implemented in child classes* - resizes rule database according to change in model size. Called by InferenceEngine::ResizeRulebase(). May clear database.
- const void * GetRule(int *index) – *implemented in child classes* - returns read-only pointer to rule identified by index array. Will return NULL iff the object is not initialized properly. Pointer to some "dummy" zero rule may be returned if selected rule is not stored in database.
- const void * GetRule(size_t num) – *implemented in child classes* - returns read-only pointer to rule identified by rule number. Will return NULL iff the object is not initialized properly. Pointer to some "dummy" zero rule may be returned if selected rule is not stored in database.
- void *GetRuleAddress(int *index) – *implemented in child classes* – returns r/w pointer to rule. Will return not-NULL iff the object is initialized and rule is stored in the database.
- void *GetRuleAddress(size_t num) – *implemented in child classes* – returns r/w pointer to rule. Will return not-NULL iff the object is initialized and rule is stored in the database.
- size_t Size() – *implemented in child classes* – returns number of rules stored in the database.
- BOOL Load(LexAnalyser &l) – *implemented in child classes* – restore database from LexicalAnalyser.
- size_t Index2Num(int *index) – converts rule index to rule number. Rules are numbered consequentally starting from 0 for (0,0,…,0), 1 for (0,0,…,1), etc.
- void Num2Index(size_t num,int *index) – converts rule number to rule index. Index array is not allocated by Num2Index.
- BOOL Decode(Chromosome &c,DWORD param) – *implemented in child classes* – decodes rule database state from chromosome.
- int **GetRulesIn() – returns pointer to array of pointers to index array, describing input parts of all rules in database. Array is Size() elements large. Each element of the array and array pointer itself should be deleted after use.
- LPVOID *GetRulesOut() – returns pointer to array of pointers to the each rule data. Array is Size() elements large. Each element points to memory area of size _rule_len which stores rule data in InferenceEngine-defined format. Each element of the array and array pointer itself should be deleted after use.
- void SetRulesOut(LPVOID *new_rules) – sets rule database state. New_rules array is in the same format as returned by GetRulesOut().
- BOOL AddRule(int *index,LPVOID *r) – *implemented in child classes* – add rule with given index to database. R points to rule data in Engine-dependent format. If rule already exists then its data is replaced with the new one.
- BOOL DeleteMF(int var,int mf) – *implemented in child classes* – remove all rules which include membership function mf of input variable var and renumber other rules.
- BOOL DeleteRule(size_t num) – *implemented in child classes* – delete rule from the database.
- BOOL HasRule(int *index,int level,void *prev,LPVOID *addr) – *implemented in child classes* – Check if rule with given index can exist in database. Index is index of the rule to search, level is number of the variable for which index is checked, prev is an address returned by this function with same index and level=level-1. Function returns TRUE if first level of indexes of index array are valid prefix for some rule in database and sets addr to value for the next call or pointer to data if level is equal to number of input variables –1.
- size_t EnumRule(size_t pos,void **rule) – *implemented in child classes* –Enum rules in the database. Pos is rule postion and should be set to 0 for the first call and incremented with each call. Rule is a pointer to pointer which receives pointer to rule data. Returned value is rule number or 0xFFFFFFFF when no more rules are available.
- size_t Index2Num(int *index) – convers rule index to rule number.

- void Num2Index(size_t num,int *index) – converts rule number to rule index. Index array should be large enough to hold rule index.

8.2.9 CompleteRuleBase Class

- Rules database implementation which stores complete database.

8.2.9.1 Base Class: RuleBase

8.2.9.2 Constructor:

- CompleteRuleBase(InferenceEngine &engine) – sets target engine to engine.

8.2.9.3 Class Variables:

- void *_data – pointer to database. Complete database is stored, starting from rule with index (0,0,…,0) then (0,0,…,1), etc.
- size_t _db_size – number of rules in the database.

8.2.9.4 Operations:

- int Resize() – resizes rule database according to change in model size. Called by InferenceEngine::ResizeRulebase(). Clears database.
- const void * GetRule(int *index) – same as GetRuleAddress, since all rules are stored.
- const void * GetRule(size_t num) – same as GetRuleAddress, since all rules are stored.
- void *GetRuleAddress(int *index) – returns r/w pointer to rule. Will return NULL iff the object is not initialized.
- void *GetRuleAddress(size_t num) – returns r/w pointer to rule. Will return NULL iff the object is not initialized.

Other functions match specifications of RuleBase class.

8.2.10 LBRWRuleBase Class

- Rules database implementation, which stores partial database. Only N rules which gets higher level of probability on given learning signal are stored.

8.2.10.1 Base Class: RuleBase

8.2.10.2 Constructor:

- LBRWRuleBase(InferenceEngine &engine) – sets target engine to engine.

8.2.10.3 Class Variables:

- size_t size - maximal number of rules to store. Should be set before Create() call.
- void *data – pointer to rule data.
- size_t * index – database index (numbers of stored rules).
- float * level – level of rule is stored here during Create() procedure.
- size_t found – number of rules actually stored in the database.
- void *empty_rule – this pointer is returned by GetRule() if requested rule is not stored in database.
- index_tree *tree – tree representing combinations of indexes of rules stored in db.

8.2.10.4 Operations:

- int Resize() – resizes rule database according to change in model size. Called by InferenceEngine::ResizeRulebase(). Clears database.

- const void * GetRule(int *index) – returns read-only pointer to rule identified by index array. Will return NULL iff the object is not initialized properly. Pointer to empty_rule will be returned if selected rule is not stored in database.
- const void * GetRule(size_t num) – returns read-only pointer to rule identified by number num. Will return NULL iff the object is not initialized properly. Pointer to empty_rule will be returned if selected rule is not stored in database.
- void *GetRuleAddress(int *index) – returns r/w pointer to rule. Will return not-NULL iff the object is initialized and rule is stored in the database.
- BOOL AddToIndexTree(size_t num, BYTE *r_data) – add rule identified by number num to index tree. R_data is a pointer to rule data.
- void FreeIndexTree(index_tree *branch=NULL) – removes index_tree element and subtree. Call with branch set to NULL to clear entire tree.
- BOOL BuildIndexTree(void) – builds new index tree from index array.

8.2.11 OutCombiner Class

- Base class for generalization algorithms.

8.2.11.1 Constructor:

- OutCombiner() – construct object.

8.2.11.2 Class Variables:

- FloatVector _result - result vector.
- int _size – number of databases.

8.2.11.3 Operations:

- void SetSize(int s,int var) – inform object about model size. S is a number of databases, var – number of output variables.
- int GetSize() – return expected number of databases.
- const char* GetName() – returns human-readable algorithm description
- void Init() – prepare object for new calculation.
- void AddVector(FloatVector &v) – process new output vector. This function is called consequently with all vectors resulting from inference.
- FloatVector Output() – returns generalization result.

8.2.12 OutSwitcher Class

- Switches between different generalization algorithms

8.2.12.1 Base Class: OutCombiner

8.2.12.2 Constructor:

- OutSwitcher() – construct object.

8.2.12.3 Class Variables:

- bool _enable_qfi – flag defining operation mode. TRUE- quantum algorithm, FALSE - averaging.
- OutAverage _avg – averaging algorithm implementation.
- OutQFI _qfi – quantum algorithm implementation.

8.2.12.4 Operations:

Reimplement OutCombiner class functions, redirecting them to one of the enclosed algorithms as set by _enable_qfi variable.

8.2.13 OutAverage Class

- Implement quantum generalization algorithm

8.2.13.1 Base Class: OutCombiner

8.2.13.2 Constructor:

- OutAverage() – construct object.

8.2.13.3 Class Variables:

- Umatrix *_ent – Entanglement matrix.
- StepMatrix *_int – Interference matrix.
- IntArray _fn – Input function.
- PCalc *_prob - probability calculators for all data
- float *_input; - input values for all bases and output variables
- float *_i_p - probability values for all bases & variables
- int _count - current db counter
- int _data_size - Number of bases * number of output variables
- int _vars - Number of output variables
- float *_v1 - Quantum vector
- float *_v2 - Quantum vector

8.2.13.4 Operations:

Reimplement OutCombiner class functions.

8.2.14 OutAverage Class

- Implement weighted average generalization algorithm

8.2.14.1 Base Class: OutCombiner

8.2.14.2 Constructor:

- OutAverage() – construct object.

8.2.14.3 Class Variables:

- FloatArray _weight – array of weights.
- int _count – number of processed vectors

8.2.14.4 Operations:

Reimplement OutCombiner class functions.

8.2.15 TextSource Class

- Base class for text source for lexical analysis. Represents stream of characters.

8.2.15.1 Class Variables:

- TCHAR Current – current character of stream.
- int pos_str – number of current string (counted from 0), will be 0 for sources that do not work with multi-string data.
- int pos_char – number of current char within string. Counted from 0.

8.2.15.2 Constructor:

- TextSource() – construct object (do nothing). Child objects should move to the first character.

8.2.15.3 Operations:

- int EOT() – *implemented in child classes* - returns TRUE if the data is over.
- int Advance() – *implemented in child classes* - move to the next character. Returns false if no more data or error.

8.2.16 StringSource Class

- Data source for LexicalAnalyser – null-terminated memory string.

8.2.16.1 BaseClass: TextSource

8.2.16.2 Class Variables:

- String s – source string.
- int idx – index of current character within string s.

8.2.16.3 Constructor:

- StringSource(const TCHAR *str) – copy string str to s and set index to 0.

8.2.16.4 Operations:

implemented according to TextSource class description.

8.2.17 StdinSource Class

- Data source for LexicalAnalyser – input from keyboard. End of data is a new line character. The data is read from keyboard untill new line is entered even in the case of error.

8.2.17.1 BaseClass: TextSource

8.2.17.2 Class Variables:

None.

8.2.17.3 Constructor:

- StdinSource() – read first character from keyboard.

8.2.17.4 Operations:

implemented according to TextSource class description.

8.2.18 FileSource Class

- Data source for LexicalAnalyser – input from keyboard. End of data is new line. The data is read from keyboard until new line is entered even in the case of error.

8.2.18.1 BaseClass: TextSource

8.2.18.2 Class Variables:

- FILE *f – input file descriptor.

8.2.18.3 Constructor:

- FileSource(FILE *f) – set in = f and read first character from file. f is not fclose()'d by the object.

8.2.18.4 Operations:

implemented according to TextSource class description.

8.2.19 LexAnalyser Class

- Lexical analyzer used to parse saved files and user input.

8.2.19.1 Class Variables:

- lex_type lexem - current lexem type. lex_type is enumerated type, possible values are: lt_eof – for end of data, lt_word – word, lt_r_br – ")", lt_l_br – "(", lt_sep – list separator (coma), lt_num – integer number, lt_float – f.p. number, lt_str – string constant, lt_r_cb – "}", lt_l_cb – "{", lt_colon – ":".
- String word - data for lt_world & lt_str type lexems.
- float num - data for lt_float lexem.
- unsigned long integer - data for lt_num lexem.
- int int_sign – sign of 'integer' variable.
- int pos_chr,pos_str - start position of the current lexem, from (0,0).
- TextSource &_src – source of input characters.

8.2.19.2 Constructor:

- LexAnalyser(TextSource & src) – sets source to src and loads first lexeme.

8.2.19.3 Operations:

- BOOL Advance() – advance to the next lexeme.
- BOOL ExpectLexem(lex_type type) – check that current lexeme type is type and advance to next lexeme. Returns FALSE if lexeme type do not match.
- BOOL ExpectNum(float &f) – check that current lexeme type is lt_float or lt_num and set f to number read.
- BOOL ExpectInt(int &f) – check that current lexeme type is lt_num and set f to number read. Adavnce to the next lexeme if type is correct and return TRUE.
- BOOL ExpectUInt(unsigned int &f) – check that current lexem type is lt_num and set f to number read. Adavnce to the next lexeme if type is correct and return TRUE.
- BOOL ExpectConstant(String &s) – check that current lexeme type is lt_word or lt_str and set s to word read. Adavnce to the next lexeme if type is correct and return TRUE.
- int ExpectWord(const TCHAR *str) – check that current lexeme type is lt_word and loaded word match one of the words specified by str. Words are separated by commas (,). On success returns 0-based index of matched word, negative value is returned in the case of failure.
- BOOL ParseWord() – read a word from source. Words are started with any symbol expect spaces, recognised separators and digits. Words end by space or separator.
- BOOL ParseNum() – reads a number from source. If FPMode is set to TRUE then floating-point numbers are parsed, if FPMode is set to FALSE – only integers.
- BOOL ParseConstant() – reads a string constant from source. String constant is enclosed with single qotation marks ('). C-style escape sequences are recognized inside constant.
- static void LoadLocale() – fills Locale structure with data from windows regional settings.
- BOOL CheckAndAdvance(TCHAR * str) – checks if input of analyzer equals to str and advances to the end of found match. Returns TRUE if input matches str. If input does not match str then FALSE is returned.

8.3 Helper classes:

8.3.1 String

– text string. Standard string operations supported.

8.3.2 Array

– template-class for dynamic arrays.

8.3.3 SparceArray

– template-class for dynamic sparce arrays.

8.3.4 FloatVector

– dynamic sparce array of floating-point numbers based on SparceArray template.

8.3.5 FloatArray

– dynamic array of floating-point numbers based on Array template.

8.3.6 SizeArray

– dynamic array of size_t elements numbers based on Array template.

8.3.7 IntArray

– dynamic array of integer numbers based on Array template.